A REAL TIME GAMING SYSTEM

by

Edward Anthony Singer

UNITED STATES
NAVAL POSTGRADUATE SCHOOL

THESIS

A REAL TIME GAMING SYSTEM

by

Edward Anthony Singer, Jr.

December 1968

A REAL TIME GAMING SYSTEM

by

Edward Anthony Singer, Jr.
Lieutenant, United States Navy
B.A., Miami University, 1961

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
December 1968

ABSTRACT

A system is proposed which will support computer gaming in real-time. This system will, when combined with the user's Control Program, monitor all of the functions necessary to provide real-time man/machine interaction with the game. The formal definition of a programming language (RTGS Control Program Commang Language) is given; this language, supplemented by Fortran IV and IBM OS/360 Assembler Language is used for coding the user's Control Program. Plans for implementation on an IBM System/360 Model 67 are discussed and a sample program is given.

TABLE OF CONTENTS

FOREWORD

The proposal made here is for a real-time system for gaming. The system encompasses a source language adaptable to gaming and a compiler and monitor system for generating and running games. The aim is to provide the capability for gaming in real-time with dynamic interaction between the players and the computing machinery. This is to be done without the requirement for tedious attention to the details of real-time programming or a special hybrid computer installation. The system is designed to run on a general purpose digital computer with a tele-processing capability.

The specific installation for which the pilot system was designed is the IBM OS/360, Model 67 installed at the Naval Postgraduate School. The system has not, at this time, been fully implemented; however, the feasibility of such a system has been investigated by partial implementation.

The thesis presents a formal definition of the system. It goes into some detail on the individual commands; however, it is not meant to serve as a users' manual for the system. Most of the details of generating and running the system were intentionally left out since they are mostly a function of the final implementation.

The present intention of the author is to continue with the implementation and provide a full scale system. It is anticipated that when the system is available, a Technical Paper will be published with the details of the implementation.

CHAPTER I

INTRODUCTION

In recent years, gaming and simulation techniques have emerged as powerful tools to aid in the solution of problems hitherto unsolvable by conventional techniques. Gaming and simulation techniques are used to model "real world" situations in order to test the sensitivity of various parametric changes. In this manner it is possible to try various courses of action and make observations as to which are best.

Programming techniques for gaming vary markedly, being mostly a function of the type of computing machinery upon which the game will be run. At one end of the scale is the completely "self-contained" game, programmed to run on a general purpose digital computer with no human intervention during the play of the game. At the other end of the scale is the game programmed for hybrid computing systems or training devices which depend very heavily on "man-machine" interaction for their operation. These broad categories of programming techniques will be discussed separately.

Self Contained Games

Self contained games do not require any intervention while they are running. In this type gaming there are two basic programming techniques employed; time-step gaming and event-store gaming.

Time-step game. The time-step game is, as its name implies, a game in which the play takes place in discrete frames of time. Each frame of time represents a cycle in the game. During each time cycle, the elements of the game are tested and decisions are made as to how they interact. The elements are then updated to represent their new positions for going into the next cycle. The clock is advanced an amount equal to the cycle time and the game continues in the same manner until some pre-determined number of cycles have been processed or until some condition occurs. As the game proceeds, various parameters may be measured, statistics collected, tests conducted, etc.; the results are recorded for outputting after the game is completed. One of the major drawbacks of this type of game is that it is essentially discrete in its representation of the "real-world" and the only way to make it appear continuous is to make the time cycle very short with respect to the factors which are being measured. If this is not done, the resolution of the results may be unacceptable. For example, in a system where minutes are important it may be necessary to use the second as the basic time unit in order to get smooth results; however, each time-cycle usually involves a complete, or at least partial, updating of all of the force-units in the game as well as all other variable factors, a large portion of which do not effect the outcome of the game within the current cycle. The result is that, in some games, much unnecessary computation and manipulation

of data is done. Consider the scenario illustrated in Figure 1 which represents a small segment of a theater-of-operations.

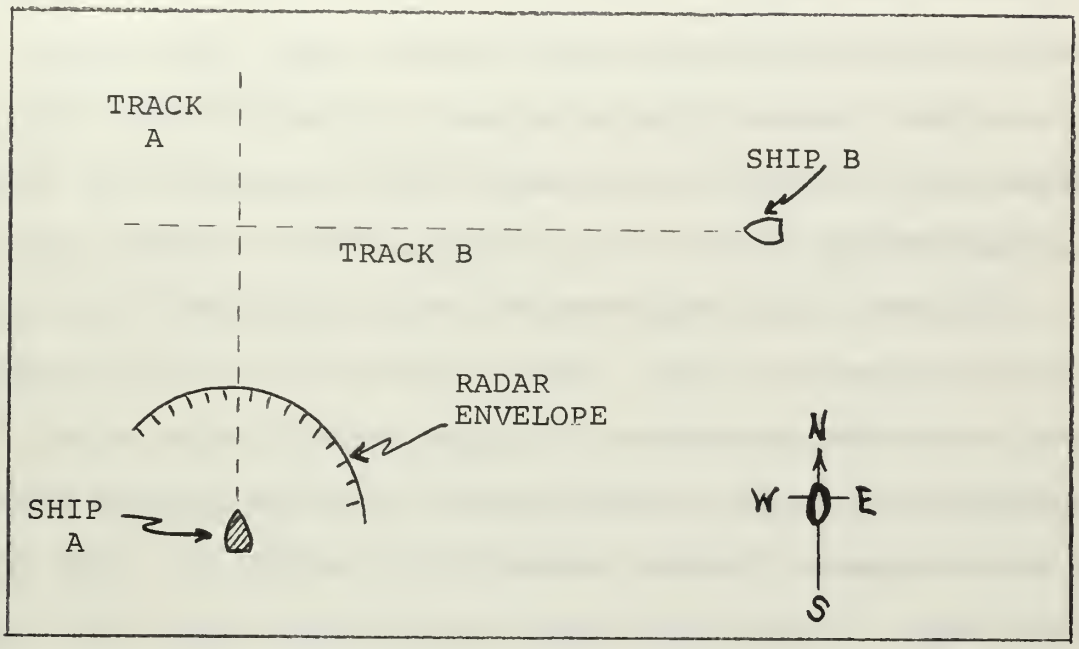


Figure 1

Ship A is searching north along track A using a radar set that detects according to some probabilistic range law. Ship B is proceeding west along track B and will eventually come into the radar envelope of ship A. It is desired that measurements of the range at which the first detection occurs and the relative positions of the ships at that time be recorded. If we assume that the maximum relative closing rate of the two ships is fifty knots and that we must have the range measurement accurate to within one hundred yards, then the longest period of time that we can allow between measurements is .01 hours or roughly thirty six seconds. Hence, if the above situation were part of a time-step game, the

time-cycle might be set at, say, thirty seconds. The worst resolution in distance that could be expected under these circumstances would be about one hundred yards, with improvement at relative closing rates smaller than fifty knots. Now assume that the two ships start at positions twenty five miles apart closing at a rate of seven knots and that the average detection range on ship A's radar is eleven miles. It will take about two hours for ship B to close to this average detection range. With a time-cycle of thirty seconds, the expected number of cycles before there is any activity on the radar is two hundred and forty plus or minus a few to account for the probabilistic effect in radar detection range. During each cycle, ship A and ship B will be advanced thirty seconds along their respective tracks. The probabilistic range law will be applied to determine if the detection has occurred, the necessary factors, if any, will be recorded and the clock will be advanced thirty seconds -- the game is now ready for the next cycle. In this particular instance the time-step procedure is not the best way to program the situation.

There are, on the other hand, games where a discrete time-step is the best way to operate. Specifically, games in which time is not an important consideration, but rather, the order in which things occur. For example, in the board game of Chess, the players alternate at making moves and time is not a factor. This game could be programmed as a

time-step game where "time" is actually a misnomer and each cycle would consist of a decision and move by one of the players.

Event-store game. Many games that involve force-units moving in a non-discrete manner can be broken up into a number of different events which occur with a varying amount of time between. For example, in the simple two-ship scenario just given there is just one event, that of making a detection.

When a game can be broken up into events occurring at some determinable time, it may be best to use an event-store type game. The primary difference between an event-store game and a time-step game is that in the former a time cycle is not of fixed length. Rather, the occurrence of specific events in the game determines the length of time between cycles. At the completion of one cycle, it is determined at what time the next significant event is to occur in the game; the clock is then advanced directly to that time and the necessary activity associated with that event is performed. Parameters of the game which are not associated with that event are not changed.

In the example given, an event could be defined, and a routine provided, to record the distance from ship A to ship B and the relative positions of the ships. Then at the beginning of the situation a mathematical computation could be performed to determine the exact time at which the detection would occur; an event could be stored for execution at the computed time. Since, in this case, that event would be

the only one stored, the clock could immediately be advanced to the detection time and the routine would be executed, recording the necessary data. In this example, considerable computational effort would have been saved.

The price that must be paid for this efficiency is usually complexity. Event-store games are considerably more complicated than time-step games.

In both of these game types, one consideration is getting the games into and out of the computing machinery as soon as possible. As soon as all the actions or events due at a certain time are accomplished, the "game-clock" is advanced either a fixed amount as in the time-step game or to the next event-time as in the case of the event-store game. The computations are performed, etc. Several replications of a game that may represent hours of real time may thus be compressed into minutes or even seconds of actual machine running time. In order to program a game of this type it is necessary to load the decisions that are to be made in all situations into the program ahead of time. This is either done in the form of mathematical functions or tables. Then, when a decision point is reached it can be immediately resolved. There is usually no "human" intervention in the game once it has begun.

Interaction Games

It may be the case that one of the factors to be measured by the game is the effect of human decision making in the face of the available information and it may not be a simple matter to characterize all of the possible conditions

ahead of time so that the resulting decisions can be tabulated. A good example of this sort of game is the game of Chess. There are so many combinations of moves and countermoves in this game that it would be infeasible, if not impossible, to store in advance the decision to be made for every situation. It may be more feasible to wait until certain situations develop and then let a "human" decision be made. Another situation where it would be useful to have "man-machine" interaction is the case where the game is being used as an aid to the teaching of decision-making.

Consider the scenario of a Naval ship such as a destroyer searching for a submarine. It is desired that certain newly proposed tactics for the surface ship be evaluated. The use of "self-contained" wargaming techniques involve programming the basic scenario with the application of the new tactics as a variable. Then the game is run several times with slight variations in the tactics; possibly with several replications on each, in order to collect statistical data on how effective the new tactics could be expected to be in the particular environment. Once the parameters of the game have been set, it is not possible to make any changes. This type of wargaming is excellent for the evaluation of new tactics; however, when used as a training device it loses much of its appeal since the players must make their decisions in a block before the run and then observe the results after the fact. There is no dynamic interaction, and hence, sometimes very little feel for what went on in all of the intermediate stages.

CHAPTER II

REAL TIME GAMING SYSTEM

The gaming system proposed here is aimed at filling the gap between strict self-contained games run purely for the evaluation of strategies and tactics and the hybrid training devices built specifically for the purpose of training people in decision-making. The class of games which this system is designed to handle are those games in which most of the decisions are made dynamically by the players in real-time. In this sort of situation, the computing machinery functions primarily as an umpire and bookkeeper, making sure that the decisions made by the individuals are legal in the framework of the game, and that all of the proper data and statistics are collected as the play of the game proceeds.

The aim of the system is to provide a real-time gaming capability for a general purpose digital computer with a minimum of effort on the part of the programmer. Specifically, the system described here is proposed for implementation on the International Business Machines System/360 Model 67 installed at the Naval Postgraduate School. The details of machine configuration are listed in Appendix H.

In a real-time system a great deal of effort must be devoted to general bookkeeping functions such as: event management, input/output, maintaining force-unit status information, and collecting data. Many of these functions are the same from one game to the next with the only significant

difference being the flow of logic associated with the various decisions in different game scenarios. The major features of the system are discussed in the following sections.

Batch processing compatability

One important attribute of a real-time system which is to be run on computing machinery normally devoted to other tasks, is that it run with a minimum of interference to those tasks. The proposed system is designed to operate in a multi-programming environment capable of processing several users' programs at the same time. In this manner, the Real Time Gaming System can operate concurrently with the normal batch processing stream. The amount of central processor time that is required by the system will usually be proportionately quite low compared with the total time that the game will be running on the system. Hence, it is to be expected that the degradation to normal batch service will be quite low.

Input/output management

The primary device for input/output for this system is the standard tele-processing remote terminal provided by the computing machinery configuration. Each of the players, as well as the umpire, is stationed at one of these terminals and it is from this unit that he receives information and makes decisions as the play of the game proceeds.

Under control of the generated Control Program, information can be freely passed back and forth among players and

between players and the system. This information includes message traffic, questions and replies, and control information to maneuver one's forces.

Control Program Command Language

In order to provide the functions required by the game, the control program must be coded by the user. In it the user provides all of the programming required to handle the specific circumstances which arise during the play of the game. The coding for the Control Program can be a mixture of some or all of the following:

1. RTGS Control Program Command Language
2. OS/360 Assembler Language[1]
3. Fortran IV[2]

The command language provides most of the facilities that are required in a gaming environment. It will be discussed in much detail later. The capability to use Fortran IV source programming has been provided so that mathematical calculations may be performed easily and the Assembler Language capability has been provided to handle logical information, string manipulation, shifting, etc.

---

[1]IBM Corporation, IBM System/360 Operating System Assembler Language, Form C28-6514 (Poughkeepsie: IBM Corporation, 1964)

[2]American Standards Association, FORTRAN, Communications of the ACM, Vol. VII, No. 10 (Baltimore: Association for Computing Machinery, 1964)

Status-of-Forces Management

As the play of the game proceeds, various force-units are introduced to and deleted from the theater of operations. They may also have various of their attributes such as velocity or position undergoing constant revision.

The Status-of-Forces Monitor is provided to automatically keep track of force-units and keep the information current and correct. The user has the capability of adding, changing, reading out, or deleting any force information dynamically during the play of the game.

Event Management

The proposed gaming system is basically an event-store type game with procedural modifications in order to accommodate the real-time feature. All event management is done automatically by the system. The control program can generate events dynamically and turn them over to the system which, in turn, ensures that their corresponding event routines are executed when they come due. The event routine is provided by the user. It is the actual sequence of operations that is to take place when an event occurs.

Time Management

Due to the real-time nature of the game, quite a bit of consideration must be given to keeping track of time. The basic time unit of the game is one second. Game-time is measured in true time since the beginning of the game; hence, the game-time at the end of one hour of play will be 3600. The maximum game-time that the system can accommodate

is 65535 seconds, which is eighteen hours, twelve minutes and fifteen seconds. If this game-time is ever reached the game will terminate.

It is possible to add a base-time to the game-time to make it correspond to any desired time-of-the-day in order to add realism. Most readouts are displayed in hours, minutes, and seconds. Several timing features are available to the generated Control Program and the timing services are all provided automatically by the system.

Statistics Gathering Services

As the play of the game proceeds, it is possible to gather any desired statistics in a running log or to tabulate key parameters in tables and/or counters. This data is available at the end of the game for offline processing.

CHAPTER III

SYSTEM OVERVIEW

The Real Time Gaming System consists of three major subsystems. The Monitor System, the generated Control Program, and the Support Program Package.

The Monitor System

The monitor system is a fixed segment of the system which changes very little from one game to the next. It controls all input/output functions, message handling, timing functions, and multi-programming linkages. The game is always under positive control of the monitor system and it is through the monitor system that all linkages are performed. At the time of system generation (to be discussed later) the monitor will be built to accommodate the system size that is desired -- this is mostly a function of how many players the system is to manage. At the heart of the monitor system is the Conversational Terminal Access Method (CTAM)[3] support package which performs the complicated input/output functions of the remote tele-processing terminals. The Input/Output Control Routine (IOCR) maintains an Input/Output Queue Control Block (IOQCB) for each of the remote terminals that may be on the line. This block contains the number of messages that are in the queue to be sent to each terminal, the number of outstanding replies, and a pointer to the first Input/Output Control Block (IOCB). An

---

[3]Columbia University, Conversational Terminal Access Method (CTAM), Columbia University Computer Center report CUCC-R-13 (New York: Columbia University, 1967)

IOCB is dynamically created each time a message must be sent to a terminal. It contains such information as: the length of the message, the actual text of the message, whether a reply is required and if so what to do with it, the time the message was sent, and other control information. The IOCB is placed in a chain as it is created with each IOCB pointing to the next. The storage requirement necessary for an IOCB varies with the length of the text of the message; however, it is dynamically allocated when required and released when no longer needed. A more detailed description of message handling is given in Chapter X.

The monitor system performs all time functions. All timing information comes from the digital timer provided by the computer operating system. The monitor system sets, and processes all timer interrupts required to service the real-time requirements of the game. When control is relinquished to the other jobs in the batch processing stream, the timer is set to provide an interrupt causing a return to the gaming system when the next function (such as an event) comes due.

The Status-of Forces Monitor segment of the system performs the necessary functions to keep the status-of-force information for the individual force-units up to date. It performs the updating functions when a reference is made to a specific force-unit. Status information is stored in Force Status Control Blocks (FSCB) which are dynamically created when new force-units are added to the system. These blocks are chained as they are created with the Status-of Forces

Monitor maintaining the pointers necessary to retrieve the information. When a force-unit is deleted from the system, the block is removed from the chain and the storage is released. A more detailed explanation of the Status-of-Forces Monitor is given in Chapter V.

The Event Queue management section of the monitor system performs the functions required to place the events in a queue and set the appropriate flags when events are due to be executed. An Event Queue Control Block (EQCB) contains information on queue size and a pointer to the first Event Control Block (ECB) as well as the "event-due" flag. The ECB contains the actual information required to execute the event routine at the correct time. It is created dynamically in much the same manner as the FSCB and the IOCB. A detailed explanation of event queue management is given in Chapter VI.

The Control Program

The flexible part of the Real Time Gaming System, and the part that makes it available to a wide variety of users, is the Control Program.

The Control Program consists of a sequence of commands which are provided by the user to handle all of the situations that can arise in the play of the game. The commands are primarily taken from the RTGS Control Program Command Language supplemented by OS/360 Assembler Language and Fortran IV[4] source coding, if desired. The command language

---

[4]IBM Corporation, IBM SYSTEM/360 FORTRAN IV Language, Form C28-6515 (Poughkeepsie: International Business Machines Corporation, 1966)

resembles any other higher level computer language in form but in substance the commands provided are specifically tailored to gaming requirements. In this manner it is possible to perform complicated gaming functions by issuing single commands. The structure of the command language is the subject of Chapter IV.

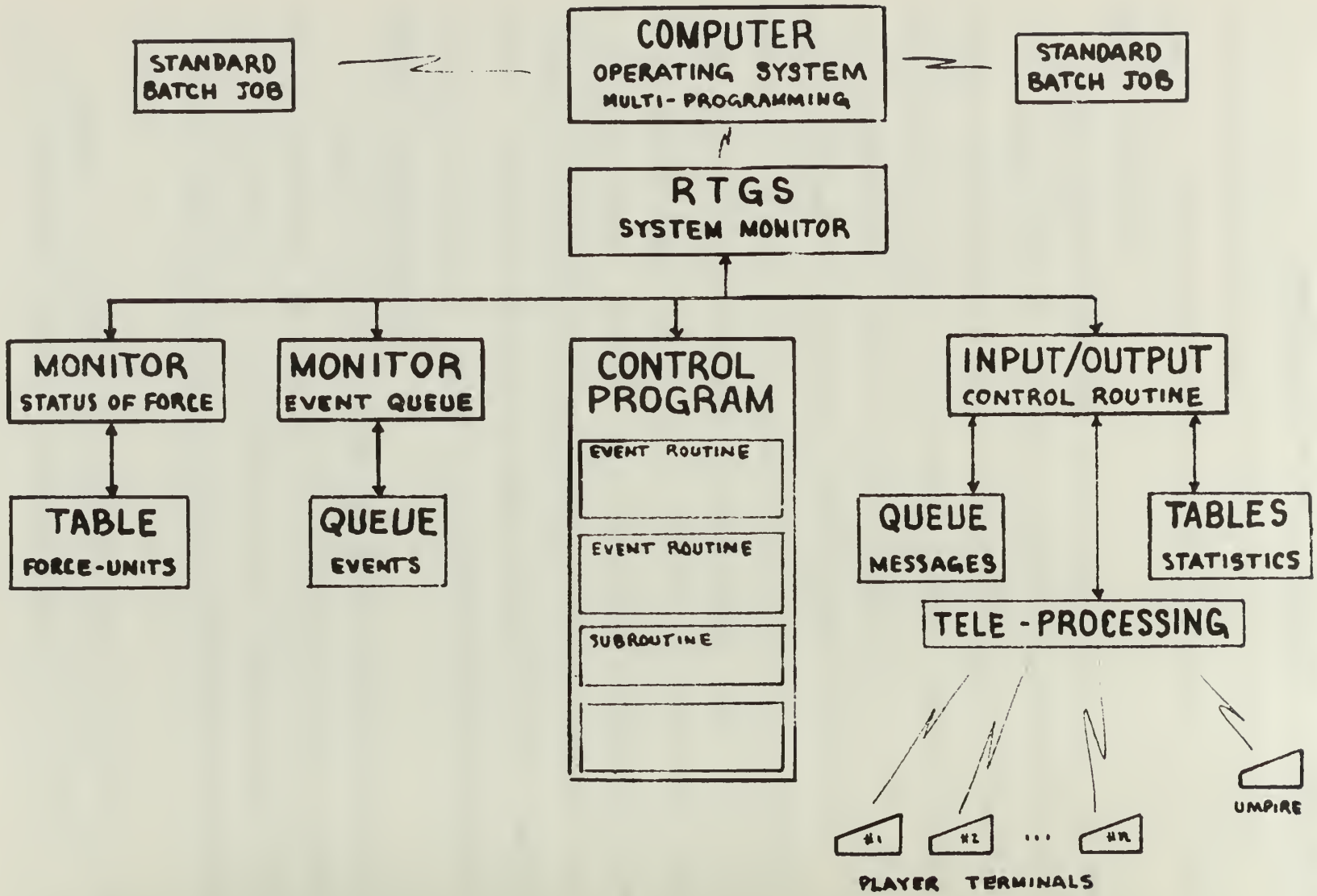
Coding must be supplied at the entry point for the game. It is usually used to initialize certain variables and to perform other bookkeeping functions. A section of coding must then be provided to handle each event. When an event is to be executed, control will be transferred to this section of coding. In addition, miscellaneous sections of coding can be provided for execution at any time. Subroutines written in any language and compiled into object form can be edited into the system and called by the Control Program when needed. Fortran IV and Assembler Language source coding may be inserted in the control program wherever desired. A special command has been provided in the command language to separate coding in different source languages.

Support Program Package

A support program package has been provided for such routine functions as fixed-point to floating-point conversion, computation of square root, etc. These modules will be edited into the overall system at generation time as required.

Figure 2 represents the overall flow of activity of the system within the computer operating system environment

during the play of the game. Note that the system monitor is the only point of linkage between the Real Time Gaming System and the computer's operating system.


STANDARD BATCH JOB
COMPUTER
OPERATING SYSTEM
MULTI-PROGRAMMING
STANDARD BATCH JOB
RTGS
SYSTEM MONITOR
MONITOR
STATUS OF FORCE
MONITOR
EVENT QUEUE
CONTROL PROGRAM
EVENT ROUTINE
EVENT ROUTINE
SUBROUTINE
INPUT/OUTPUT
CONTROL ROUTINE
TABLE
FORCE-UNITS
QUEUE
EVENTS
QUEUE
MESSAGES
TABLES
STATISTICS
TELE-PROCESSING
#1
#2
...
#N
UMPIRE
PLAYER TERMINALS

CHAPTER IV

RTGS CONTROL PROGRAM COMMAND LANGUAGE

The Real Time Gaming System is essentially a macro-compiler making use of the Assembler Macro Language[5] to as great a degree as possible. The command language consists of a series of macros which expand into Assembler Language coding during the system generation phase.

Definitions

Some definitions which will be required for an understanding of the command language are given here:

Symbolic reference. A symbolic reference is a word having up to eight alphameric characters beginning with a letter or the special character #. They are used for two purposes:

1. Branch point or entry point names: A name used to identify the location of a block of coding so that some other section of the program may branch to it. Symbolic references of this type may not start with the special character #.

2. Variable names: A name used to identify the storage location of a variable. Care must be taken not to confuse variable names with the actual variables. If the variable is fixed-point it will be referenced by a name starting

---

[5] IBM Corporation, IBM SYSTEM/360 Operating System Assembler Language, Form C28-6514 (Poughkeepsie: IBM Corporation, 1964), pp. 59-99.

with a letter. If it is floating-point the name must start with the special character # followed by at least one letter.

Symbolic references, either branch points or variable names, must have a total of from one to eight characters and may not contain any spaces. Those variable names which are going to be used in Fortran IV source statements must contain no more than six characters in order to conform to Fortran IV conventions. The following are valid symbolic references:

BRP1	#VARIABL
BRANCH1	#V1
VARIABLE	V1

Note: In the above examples V1 and #V1 are different variables.

Command. A command is some specific instruction, or set of instructions, in the Control Program which will cause some pre-defined sequence of operations to occur. It will appear in the Control Program in one of two forms:

1. The long form of the command will appear in the format; <COMMAND WORD>. It contains more than one word in some cases and the corner brackets <> must be coded.

2. The short form of the command will appear as one to four letters coded with no spaces and without the corner brackets, e.g., CW.

Operand. Each command, except a very few, must have some modifying information. This modifying information is supplied along with a command in the operand list which may have one or more elements. The operands in the operand list

are separated by commas and the general rule is that if an operand is to be left out, the comma must still be supplied to show its absence. In the case where operands are omitted from the end of the list, and no confusion can arise, the trailing commas are not required. Operands may be of any of the following forms:

1. Fixed-point symbolic: The operand points to the storage location where the actual parameter resides. A fixed-point symbolic operand must begin with a letter and consists of up to eight alphameric characters with no spaces.

2. Floating-point symbolic: The operand points to the storage location where the actual parameter resides. A floating-point symbolic operand must begin with the special character # followed by at least one letter and consists of up to eight alphameric characters with no spaces.

3. Fixed-point constants: The operand is the actual parameter and it is represented by putting the actual number in the list. It is differentiated from the symbolic operand by not starting with a letter.

4. Floating-point constant: The operand is the actual parameter and it is represented by putting the actual number in the list preceded by the special character #.

5. Literal constant: A literal constant is a string of alphameric characters which are to be used in a message or for some similar use. It appears as the desired string enclosed in apostrophe quotes ('). If the apostrophe is

required within the text of the message, it may be coded as a double apostrophe('') to differentiate it from the delimiting apostrophes.

Command Language Format

The commands will be given in the following format:

NAME	COMMAND	OPERANDS
[name]	{<COMMAND> / C}	operand1[,{2 / 1}, operand3, ...]
WHERE		
operand1 = ··· etc.		

The following notational conventions will be used:

Small letters. Small letters in the command descriptions represent words, names, etc. that must be replaced by some meaningful coding by the user.

Capital letters. Capital letters are to be coded as shown. The same applies to the following seven special characters:

) (> < # $ '

Square brackets. [] indicate that the contents are optional and need only be coded if the default is not to be taken.

Braces. {} indicate a list from which only one thing is to be selected.

Elipsis... indicates an open-ended list.

For example, in the format shown above, [name] means that the name field is optional and that the user will supply an appropriate name if desired. <COMMAND> is the long form of the command and would appear exactly as shown. The short form is C and it also would appear exactly as shown. Operandl is required and some coding must be supplied by the user. The rest of the operands are optional; however, if the second is coded it must be coded as 1 or 2. The actual length of the operand list is open-ended. The section labeled WHERE will give amplifying information concerning the limits of operands, etc.

There are several attributes and special storage locations which are common to a great many of the individual commands. They are identified by the special character $ which must precede each. Some are specific to certain commands and will be discussed in the sections giving detailed descriptions of the commands; however, those which are common to all commands are described here:

1. Twenty five temporary storage locations which can be used as work areas are provided; they may be used at any point in the coding. They are given the symbolic names: $T1, $T2,···,$T25. The advantage of using these is that the user need not define the storage himself.

2. Common areas are provided for the purpose of passing information from one section of the Control Program to another. One hundred locations are provided and they are coded: $C1, $C2,···,$C100.

3. In those commands involving the boolean switches that are available with event and status-of-forces blocks; the eight individual switches can be accessed by the symbolic names: $ES1, $ES2,···,ES8 for the event switches and $FS1, $FS2,···, $FS8 for the force-unit switches. An additional sixteen general switches are provided for access from any point in the Control Program at any time. They are accessed by the symbolic names: $GS1, $GS2,···, $GS16.

The individual commands are given in the Appendices with a detailed description of how each one works.

CHAPTER V

STATUS-OF-FORCES

The operating forces in the game environment are divided into two major categories: stationary forces and movable forces.

Stationary Forces

Stationary forces are those force-units which are not capable of moving, in time, through the theater of operations either with respect to a geographical reference or with respect to other operating forces. As the game proceeds stationary forces are capable of taking on or changing various attributes. Examples of this type of force are:

1. A supply facility with an inventory of replacement parts or materials which must be delivered to, or picked up by friendly forces. In this case the attributes could be the type and quantity of materials available or the delay of materials once ordered.

2. A manpower pool from which men or labor can be drawn when required.

3. Base facilities which might include the following:

   a. Repair facilities such as a shipyard or labor pool of mechanics.

   b. Replenishing or refueling facilities.

   c. Airports, landing fields, combat air strips, etc.

Movable Forces

Movable forces are those forces which are capable of moving throughout the theater of operations, either without

limit or in some limited fashion. Among the important attributes that such forces can have are: position with respect to some reference point, velocity in any or all coordinate directions, and any modifying information about the force itself. Some examples of this type of force are:

1. Aircraft: Located in the theater of operations by its present position in a three-dimensional coordinate system with component velocities in the three directions. Additional modifying information would be such factors as the number of passengers aboard, payload, weapons aboard, fuel remaining, etc.

2. Ship: Located similarly to the aircraft, however, in only two-dimensions.

3. Personnel: Either a single individual or a large group represented as a single unit.

The manner in which movable forces may transit through the theater of operations will vary from one type of game to the next or even within a single game. In general; however, the movement can be characterized in two board categories, discrete and continuous.

Discrete. Either the motion or the position of a force-unit may be discrete in nature. Discrete motion implies that the movement is discontinuous and often instantaneous. Similarly, discrete position implies "pigeon-hole" placement in some specific location within the grid. The scale for discrete motion is usually ordinal and distances and velocities may be meaningless quantities as in the case of chessmen on

a chess board. It is also possible that a force under discrete motion may disappear from one position and reappear someplace else either immediately, as in the case of chess, or after some delay in time, as in the case of a submarine periscope.

Continuous. Either the motion or the position of a force may be continuous. Continuous motion implies that any position or velocity within some spectrum may be assumed. The scale for measurement is usually linear with distances and velocities being meaningful.

Some examples of the positioning of movable forces are given in Table 1.

		MOTION	
		DISCRETE	CONTINUOUS
POSITIONING	CONTINUOUS	Truck mounted radar station which must be set up in order to operate.	Naval ships or aircraft moving in the theater of operations.
	DISCRETE	Chess men moving about on a chess board.	Arrival, service, and departure of customers at a stationary service facility.

Table 1

In the proposed Real Time Gaming System the movement of forces through the theater of operations is controlled by the Status-of-Forces Monitor.

The scheme used to keep the status-of-forces up to date minimizes the bookkeeping by only updating force-unit information when a reference is made to that unit. A reference occurs in any of the following circumstances:

1. The addition of a new force-unit to the theater of operations.
2. The changing of any or all attributes of an already existing force-unit.
3. Inquiry as to the present status of a force-unit.
4. Deletion of a force-unit from the theater of operations.
5. Comparison of two or more force-units.

Since there is no way that the force-unit can have any influence on the play of the game unless there is a reference made to it, it is unnecessary to update it either periodically or continuously between references. This saves time by avoiding any more bookkeeping than is essential.

The Status-of-Forces Monitor maintains a Force Status Control Block (FSCB) for every unit in the system. This block contains the essential bookkeeping information for the unit so that the status of the unit may be determined at any time. A maximum of two-hundred fifty six blocks may be maintained by the system at any one time and each block contains the following specific information:

1. Unit designation: The unit designation is a unique number in the range (0-255) which is arbitrarily assigned to each force-unit when it is first introduced into the theater of operations.

2. Type force: Information is stored indicating whether the force is stationary or movable and in the latter case whether the movement and positioning is discrete or continuous.

3. Update-time: The update-time is the game-time at which the last update was performed -- this time represents the last time at which the currently listed status information was exactly correct.

4. Force information boolean switches: Eight switches are provided which can be either set or reset by the control program to indicate various binary information about the force-unit.

5. Force velocity: In the case of continuous motion forces the velocities are tabulated in each of the three coordinate directions. For discrete motion the velocity does not apply.

6. Force position: The position of the force-unit with respect to the origin of the grid is tabulated. In the case of continuous positioning, these are floating-point numbers which describe the distance (plus or minus) from the origin along each of the three coordinate directions. In the case of discrete positioning they are fixed-point integers which indicate the "slot" in each of the three dimensions.

7. Force attributes: Depending on the force type, up to nine attributes may be stored in the block at any time. The attributes represent variable information which the Control Program requires for processing the force-units. These attributes are set and changed only by the Control Program.

When a reference is made to an FSCB, the current time is fetched from the master clock and converted to game-time. This time is then compared with the "update-time" in the FSCB. The difference in the times represents the amount of time that has lapsed since the last update was performed. If applicable, this time is multiplied by the velocity components to determine the distances traveled since the last update. This distance is then added to the position thus giving the new position of the force-unit in the grid. The update time is then corrected and the reference is serviced. In the case of discrete motion units, it is not necessary to compute the new position as a function of time; the force-unit must be moved in discrete steps by the Control Program. The actual construction of the FSCB is given in Appendix A.

Status-of-Forces Commands

The status-of-forces commands are provided to initially insert, change, and delete force-units to or from the theater of operations. A detailed description of each of the commands is given in Appendix B; a brief summary is given here.

NEW FORCE. The "New Force" command is used to introduce a new force-unit into the theater of operations. When it is introduced by this command, the type, stationary or movable, is determined and may not be adjusted during the play of the game. Every force-unit is identified by an integer in the range (0-255) and if a new unit is introduced having the same designation as one which already is defined, the original will be cancelled.

DELETE FORCE. The "Delete Force" command is used to remove a force-unit from play. After this command has been issued, the designation number is free for use to identify another force-unit.

READ FORCE. The "Read Force" command is used to fetch the present status of a force-unit. As a result of issuing this command the force-unit's FSCB will be updated and the requested status information will be returned.

CHANGE FORCE. The "Change Force" command is used to change any or all of the characteristics of a force-unit. By using this command it is possible to go into the FSCB and change whatever factors are required in order to adjust position, velocity, etc., of a force-unit.

DISTANCE BETWEEN. The "Distance Between" command can be used to compute the distance between two force units in any one or combination of several coordinate directions.

SET SWITCH ON. The "Set Switch On" command will set any combination of boolean switches associated with a particular force-unit to the ON position. This allows decision making information to be stored in the FSCB for a unit.

SET SWITCH OFF. The "Set Switch Off" command operates in the same manner as the "Set Switch On" command; however, it places the switches in the OFF position.

CHANGE SWITCH. The "Change Switch" command operates in the same manner as the "Set Switch On" or the "Set Switch Off" commands; however, it changes the position of the designated switches to the opposite position.

CHAPTER VI

EVENT QUEUE MANAGEMENT

Throughout the play of the game, decisions are made which do not necessarily require action to be taken immediately. These decisions must be placed in a queue so that they can take effect at the proper time.

An event is any action or sequence of actions which occur at some specified time. Events can take many forms ranging from the simple posting of a flag to a long sequence of commands. There are two major categories of events in the system: those which are defined in the generated Control Program ("game" events) and those which are used by the Monitor System for control purposes ("system" events).

Game Generated Events

The Control Program may define any sequence of commands as an event; this sequence will then be executed every time that particular event comes due.

A game generated event is the means by which the Control Program can execute some specific sequence of commands when and under whatever condition it desires. When it becomes apparent that some action is due in the future, an event is placed in the event queue with specific information that will allow it to be executed at the correct time. There must be a section of coding in the Control Program to correspond to the required functions. When the event works its way to the head of the event queue at the indicated time, the Control Program will transfer control to the associated event

routine. If two events are placed in the event queue with the same execution time, the event designation will determine which event occurs first, with the lower numbered event being given priority. If both time and event number are identical, the order in which they were placed in the queue will govern -- they will be executed in FIFO order.

System Generated Events

When the system meets a requirement to execute some action in the future, it will store a "system" event in the same queue with the "game" events. A "system" event for the same time always takes precedence over a "game" event. The mechanics of "system" events is basically the same as that for "game" events; hence, they will not be discussed separately.

Event Generation and Execution

When the flow of the Control Program encounters the requirement for the generation of an event, the sequence of activities is as follows:

An Event Control Block (ECB) is constructed containing the information necessary to execute the event when it comes due. The ECB is transferred to the Event Queue Control Routine. The routine scans the queue to find the appropriate position for the newly created ECB. The block is then inserted in the queue at the proper place. Meanwhile, the regular flow within the Control Program has been interrupted by the necessity to store this event. As soon as the event has been placed in the queue, the Control Program resumes -- then at some later time (as short as a fraction of a second

and as long as eighteen hours later) the event itself will come due. When this happens the system will set a flag indicating that an event is due. Then at the first opportunity to do so, the system will shift control to the event routine which is either a Control Program defined sequence of commands or a segment of system programming, depending on whether the event was a "game" event or a "system" event.

Event Routine Construction

An event routine must be uniquely defined for every event designation which is used by the generated Control Program. It consists of a sequence of commands of any length desired; the only requirement is that the event does not loop infinitely, but rather, eventually takes an exit. Every time an event in the queue with this designation comes due, the section of coding will be executed. The modifying information which is contained in the Event Control Block is available to the event routine throughout its execution.

Event Queue Control Block

The Event Queue Control Routine maintains the Event Queue Control Block (EQCB) in order to keep track of the chaining information necessary for scanning the queue and posting the event in the queue. The block contains the following specific information:

1. Event-due flag: The "event-due" flag is set by the monitor system when the top event in the queue is ready to be executed.

2. Queue element count: A count is maintained showing the number of events that are presently stored in the queue.

3. Pointer: A pointer in the block is used to start the chaining operation when the queue must be scanned.

The actual construction of the EQCB is given in Appendix A.

Event Control Block

The Event Control Block (ECB) is generated dynamically when an event is to be placed in the queue. It is then transferred to the Event Queue Control Routine to be placed in the queue. The block contains the following specific information:

1. Event-type flag: This flag is used to distinguish "game" events from "system" events.

2. End-of-queue flag: This flag stops the chaining operation and indicates that this block is the final one in the queue.

3. Pointer: A pointer to the next block in the queue is maintained for chaining.

4. Time event due: The actual game-time that this event is due for execution.

5. Event designation: The identification of the event routine that is to be executed when this event comes due.

6. Boolean switches: Eight switches are provided which can be either set or reset by the Control Program to indicate various binary information needed for the execution of an event.

7. Event modifiers: One fullword and two halfword modifiers are provided in order to pass information to the event routine at execution time. Modifiers number one and two are halfwords and number three is fullword.

Flow of Event Activity

Figure 3 illustrates the flow of activities during the generation and execution of events. The Control Program generates an event designated as number one due for execution at time six. The event is stored in the queue and control is returned to the Control Program. At time six, the system sets a flag to indicate that an event is due. At this time the event has moved to the top of the queue. The Control Program is allowed to execute until the first opportunity comes along for it to allow the execution of an event routine. When this happens, control is transferred to the event routine for events designated one. The routine is completely executed and control is returned to the Control Program.

Example of the Use of Events

Suppose that in the play of the game it becomes necessary for a ship to fire a surface-to-air guided missile at an enemy aircraft. It is determined that the fire control solution requires the missile to be fired at game-time 3361. The Control Program could store a "fire-missile" event in the queue for time 3361. The "fire-missile" event could then be programmed to compute the time of intercept, the probabilities involved, etc. The event could then itself generate more events such as:

FLOW OF ACTIVITY IN EVENT MANAGEMENT

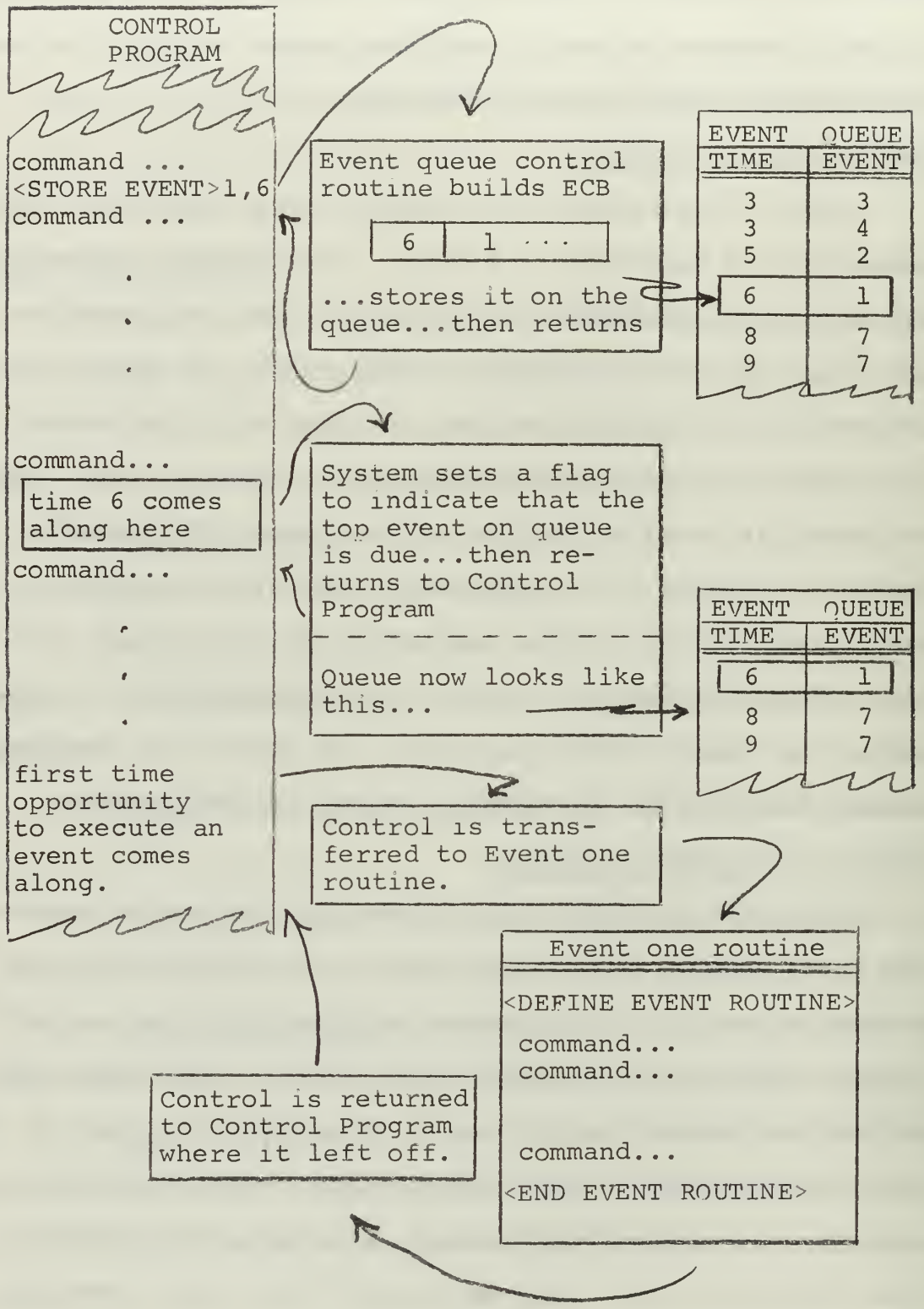


Figure 3.

1. "intercept" which could inflict damage on the aircraft under certain probabilistic constraints at a time computed by the generating event -- say 3375.

2. "reload-launcher" which could bring another missile out of the magazine when the first has had time to clear the launcher -- say at time 3369.

3. "radar-contact" which could generate a radar contact on the aircraft's detection equipment corresponding to its detection of the missile, etc.

Any of these events could then generate more events. For example, the "radar-contact" event could cause evasive action which could be determined to be successful or not by some probability distribution. Assuming it was successful it could then cancel the "intercept" event and generate an "evasive action taken" event which, in turn, could start the whole process over again back on the ship.

Event Management Commands

The event management commands are provided to define event routines and to store and cancel events. A detailed description of each of the commands is given in Appendix C; a brief summary is given here.

Define Event Routine. The "Define Event Routine" command is used at the head of each block of coding, in the Control Program, that represents the sequence of commands to be executed as the event routine. Every event routine is identified by an integer in the range (0-255) and the

corresponding identification is placed with the events in the event queue. No two event routines may have the same event designation.

End Event Routine. The "End Event Routine" command is used to mark the last command in the event routine. It also serves as the command that causes control to be returned to the Control Program when the event routine is finished executing.

Event Exit. The "Event Exit" command is provided to give the means of returning from the event routine to the Control Program in the middle of the event routine. Any number of exits may be used within an event routine.

Store Event. The "Store Event" command is used at any time in the Control Program that it is desired to designate an event for execution at some time in the future. The "Store Event" command may also be coded inside another event routine.

Execute Event. The "Execute Event" command is provided to allow the immediate execution of an event routine. No event is stored in the queue as a result of this command. When the event routine completes execution, control is returned to the next command in the Control Program sequence.

Cancel Event. The "Cancel Event" command is provided to allow cancellation of an event which has been stored in the queue. Only the first event encountered with the specifications for cancellation will actually be canceled.

Global Event Cancel. The "Global Event Cancel" command is provided as a means of canceling all of the events that are stored in a given category.

CHAPTER VII

CONTROL PROGRAM LINKAGE

In order to provide a high degree of programming flexibility, the capability exists in the Control Program to shift from one programming language to another. During the pre-processing stage in system generation, the coding from different languages can be removed and compiled as independent subroutines. The linkages which are required to pass control back and forth between routines are standardized so that no interface problems exist. The convention used is that of OS/360 Subroutine Linkages[6].

Programming may be done in any source language desired, so long as the standard conventions are followed. In cases other than Assembler Language and Fortran IV, the compilation is done separately and the object modules are edited[7] into the system at system generation time. When it is necessary to use Fortran IV or Assembler Language, source statements are inserted in the Control Program where desired, using the appropriate commands to mark the boundaries.

---

[6]IBM Corporation, IBM SYSTEM/360 Operating System Supervisor and Data Management Services, Form C28-6646 (Poughkeepsie: IBM Corporation, 1967), pp. 9-13.

[7]IBM Corporation, IBM SYSTEM/360 Operating System Linkage Editor, Form C28-6538 (Poughkeepsie: IBM Corporation, 1966).

Linkage Commands

The linkage commands are provided to ensure the proper interface exists between coding of various types. A detailed description of each of the commands is given in Appendix D; a brief summary is given here.

Assembly Language. The "Assembly Language" command is issued to indicate the coding that follows will be Assembler Language. The OS/ 360 Assembler Language is then inserted in the Control Program coding in standard format. Compatibility with the names and labels which have been assigned must be maintained. All general-purpose and floating-point registers are available for use.

End Assembly Language. The "End Assembly Language" command is inserted after the last Assembler Language source statement to indicate that the coding will return to RTGS Command Language.

Fortran. The "Fortran" command is issued to indicate that the coding that follows will be Fortran IV source statements. It is necessary to list in the operand list those variables, from the Control Program, which are to be used in the Fortran IV coding. Any such variables that are not in the list, and have the same symbolic references as variables in the Control Program, will be considered independent for purposes of this block of Fortran IV coding only. The only difference between the Fortran IV source coding used here and conventional Fortran IV source coding is that floating-point numbers are represented symbolically by starting with the special character #; whereas, fixed-point

symbolic references may start with any letter in the range (A-Z). The Fortran IV "IMPLICIT" statement may not be used in the Fortran IV coding; however, explicit type declaration statements may be used. Symbolic references in Fortran IV statements may contain only six characters in order to conform to Fortran IV conventions.

End Fortran. The "End Fortran" command serves the same function in Fortran IV source coding as the "End Assembly Language" command did.

Subroutine. The "Subroutine" command is used to indicate that the coding which follows is a subroutine coded in RTGS Command Language. The subroutine is usually used when several event routines and/or other blocks of coding perform the same function. The subroutine should not contain any entry points except at the beginning via the "Subroutine" command. The following commands should not appear in a subroutine: "Do Periodically", "Signal At Time", "Signal After Lapse", "End of Game", "Return to Monitor", or any "Branch" command where the branch is external to the subroutine. In short, the subroutine must be self-contained.

Return. The "Return" command is used to allow exit from the subroutine at any point desired within the routine. It provides for multiple exits.

End Subroutine. The "End Subroutine" command must be the last command in the coding for a subroutine and fulfills the functions of the "Return" command as well as indicating the end of the subroutine coding.

Execute Subroutine. The "Execute Subroutine" command, when executed by the Control Program, causes a branch to be taken to the subroutine named in the operand. A parameter list is also provided in the operand and it will be passed, using standard linkage conventions, to the subroutine. The subroutine must either have been defined by the "Subroutine" command or have been provided at system generation time in object form. Upon completion of the execution of the subroutine, control will proceed to the next command in the Control Program sequence.

Return Code Branch. The "Return Code Branch" command is provided to allow standard register fifteen return code conventions to be used in subroutines.

Return to Monitor. The "Return to Monitor" command is provided as the standard end to a logical block of coding, e.g., the coding which is influenced by the "Do Periodically" command or the "Signal at Time" command. When this command is encountered in the coding stream, control passes from the Control Program back to the system monitor. The next command in the coding stream must have a name field in order to provide a means for that section of coding to be executed. The "Return to Monitor" command may be used in an event routine; however, it is not used as the standard exit from an event routine -- the "End Event Routine" command and "Event Exit" command have been provided for this purpose.

End of Game. The "End of Game" command causes the game to stop. As a result of this command, the log will be brought

up to date, a tabulation will be made of events remaining in the event queue for execution, the terminals will be disabled, and control will be transferred to the computer's operating system to terminate the run.

Random Variable. The "Random Variable" command will link to a random number generator, as designated in the operand, and place the value of the random variable in a specified location.

CHAPTER VIII

TIME MANAGEMENT

The proposed system is designed to run in a real-time mode on computing machinery which normally operates in a batch-processing mode. The time-management segment of the monitor system performs the functions necessary to run the game in real-time. Several definitions are required for an understanding of how the system accomplishes the real-time management.

Real-Time

Real-time refers to the actual time-of-the-day by the clock. It effectively never stops. The basis for determination of real-time is the digital timer which the computing machinery hardware provides. In the System/360, Model 67, it is a register which is updated every twenty-six microseconds. In the mode of operation used by the RTGS monitor system, real-time is available to the nearest hundredth of a second. Internally the time is used to this accuracy to resolve some problems such as which of two "almost simultaneous" events is to occur first; however, for most purposes the time will be truncated to the nearest second.

Game-Time

Game-time is the amount of real-time which has lapsed since the beginning of the game while the game "master clock" is running. It is the basis upon which all timing in the game is controlled. Game-time is specified in seconds since the beginning of the game. It is always expressed as an

integer in the range (0,65535). This corresponds to a maximum of eighteen hours, twelve minutes and fifteen seconds representing the longest time that any game can run.

Clock-Time

Clock-time is game-time with some base added to shift it in time to some desired time-of-the-day. Clock-time is expressed in terms of hours, minutes and seconds; it is used only to add realism to the game and is never used internally for any control purposes. Most displays to the players are represented in clock-time.

Master-Clock

The master-clock for the system will always carry game-time. It may be started and stopped at any time under the control of the Control Program. The value of the master clock is available to the Control Program at any time.

Time Management Commands

The commands provided for time management are the means by which the Control Program may perform various activities which are a function of time. A detailed description of each of the commands is given in Appendix E; a brief summary is given here.

Set Time. The "Set Time" command is provided so that the Control Program can, at any time, reset the time presently in the master clock. The game-time that is desired is coded with the command. As a result of issuing this command the master clock will be stopped.

Start Time. When the "Start Time" command is issued, the master clock will be started with whatever game-time it now contains.

Stop Time. The "Stop Time" command will cause the master clock to stop with its present value. The actual time in the clock will not be adjusted. Care must be exercised when using this command. If the command which will be used to restart the clock is time-dependent (such as a command in an event routine), it will never be executed since all future events will be frozen until the master clock is restarted.

Present Time. The "Present Time" command will fetch the present game-time from the master clock and return the value as an integer in the range (0,65535).

Present Clock. The "Present Clock" command will fetch the present game-time and add the appropriate base to it in order to convert it to clock-time. This time will then be broken up into its hour, minute, and second components and placed in the location designated.

Set Clock. The "Set Clock" command is used to establish the base time for conversion between game-time and clock-time. Clock-time will automatically wrap-around at midnight.

Signal At Time. The "Signal At Time" command will indicate to the system monitor that control is to be started at the point in the Control Program indicated by the operand at a specific time also indicated by an operand. This command is a conditional branch which is a function of time. Any

number of signals can be set up for branch to some specific location without mutual interference.

Signal After Lapse. The "Signal After Lapse" command is basically the same as the "Signal At Time" command except the branch occurs after a specific time lapse rather than at a specific time.

Delay Until. The "Delay Until" command will cause execution of the commands in the Control Program stream to cease until the designated time. Meanwhile, the Control Program will execute any unexecuted event routines, and do any other work which may be backlogged. When the indicated time comes due, the Control Program will continue execution again with the next command in the sequence. When using this command care must be exercised to keep track of the values of variables. There is nothing to prevent the value of a variable which has been set prior to the "Delay Until" command from being changed by some other section of coding while the delay is in effect.

Delay For. The "Delay For" command is basically the same as the "Delay Until" command except that the delay is for some specific length of time rather than until some specific time.

Do Periodically. It may become desirable to continuously execute a section of coding in the control program in an iterative manner. The "Do Periodically" command will commence its operation at a time specified in one of the operands. The coding that follows the "Do Periodically" command

will be executed repeatedly in accordance with a cycle time specified in one of the operands. The operation will continue until either a specified number of iterations have occurred or until a time limit is reached. If many of these commands are going on at once, the system will be burdened with quite a bit of bookkeeping work and it is possible that the game will draw back in time and start running late.

CHAPTER IX

CONTROL PROGRAM LOGIC AND ARITHMETIC

During the execution of the Control Program, it may be necessary to make decisions dynamically based on the value of variables, the setting of boolean switches, or some other factor. The sequence of commands in the Control Program can be modified based on decision-making information and in this manner the play of the game can be controlled.

Branching

Most decision-making is done by branching. The sequence of commands in the Control Program is interrupted and directed to some other place in the coding. The new entry point is usually designated by a symbolic reference in the name field. This reference is then used in the operand of that command which causes the branch to occur. The basis for a branch is usually either the setting of a boolean switch or the arithmetical value of a constant.

Boolean Switches

Each event has a set of eight switches which it can pass to the event routine. In addition, each force-unit is provided with eight switches. The Control Program itself has sixteen. These switches can, independently, or in groups, be set "ON" or "OFF". Then at any time in the game they may be interrogated to cause a branch based on their setting.

Arithmetic

A limited amount of arithmetic can be performed in the Control Program using the RTGS Command Language. If complex

mathematical functions are to be evaluated, Fortran source coding should be inserted.

Logic and Arithmetic Commands

A detailed description of each of the commands is given in Appendix F; a brief summary is given here.

Set Switch On. The "Set Switch On" command can be used to turn any of the boolean switches to the "ON" position. They may be turned "ON" in groups within the same set or individually.

Set Switch Off. The "Set Switch Off" command can be used to turn any of the boolean switches to the "OFF" position.

Change Switch. The "Change Switch" command can be used to change the current position of the boolean switches to their complement.

No Operation. The "No Operation" command does nothing. It is provided as a place to attach a branch point name or to provide a place for coding that is to be added in the future.

Branch. The "Branch" command causes control to be transferred to some other section of the Control Program. It names, in the operand field, a symbolic reference to the section of coding which is to be executed next.

Branch On Value. The "Branch On Value" command will evaluate the present value of a variable and take a branch based on any of the three mutually exclusive conditions -- negative, zero, and positive.

Branch All On. The "Branch All On" command will look at the present setting of a collection of switches designated in the operand and branch if all of the switches are in the "ON" position. Similar commands are provided to branch on other combinations; specifically, "Branch Any On", "Branch All Off", and "Branch Any Off".

Add. The "Add" command will add any number of variables listed in the operand and place the result in a specified location.

Subtract. The "Subtract" command will subtract two variables and place the value in a specified location.

Multiply. The "Multiply" command will multiply any number of variables listed in the operand and place the result in a specified location.

Divide. The "Divide" command will divide two variables and place the quotient and remainder in specified locations.

Set Value. The "Set Value" command will set the value of any variable to the amount specified in the operand.

CHAPTER X

MESSAGE MANAGEMENT

All messages generated during the play of the game -- those from the system to the players and those between the individual players -- are handled in the same manner. An overview of the message handling blocks was given in Chapter III; the blocks will be described in more detail here.

Input/Output Queue Control Block

One IOQCB is established at system generation time for each of the players which will be using the system plus one extra for the umpire. It contains the following information:

1. Message count. This is a count of the number of messages which are waiting to be sent to a particular player.
2. Reply count. This is a count of the number of messages which have been sent and are presently awaiting a reply.
3. Queue-status flag. This is a flag indicating that there are presently no input/output control blocks in the queue.
4. Pointer. This is an address pointer to the top IOCB in the queue.

Input/Output Control Block

One IOCB is created dynamically every time a message must be sent to a player. If no reply is required, the storage allocated for the block is released whenever the message is sent. If, however, the message requires a reply, the block is maintained in the queue until the reply is received.

When a reply is required for a message, the message is assigned a serial number which appears with the message when sent; this serial number must then precede the reply. In this manner the reply can be matched to the correct message in those cases where more than one reply is outstanding. The construction of the IOCB is shown in Appendix A. A summary of its contents is given here:

1. Reply flag. This flag is set to indicate that the message associated with this block requires a reply.

2. Reply serial number. The field is only coded when a reply is required and it stores the serial number, in the range (0,9), associated with the message.

3. Length of IOCB. This field is used for storage allocation purposes. It contains the length of the IOCB.

4. End-of-queue flag. This flag is set when this IOCB is the last one in the queue. This information is used for chaining purposes.

5. Pointer. This field points to the next IOCB in the chain.

6. Length key. This field is coded with the length, in characters, of the message.

7. Time sent. This field indicates the game-time that the message was actually sent to the player.

8. Offset time. This field indicates the time which the action associated with a reply is to be delayed. This delay is coded in the Control Program by the command which creates the IOCB.

9. Reply event. This field is coded with the designation of the event routine, which is to be executed when the reply is received.

10. Boolean switch. This field contains the boolean switch settings to be passed to the event routine when the reply is received.

11. Text. The remainder of the block constitutes the actual text of the message.

Message Handling Commands

The message handling commands are provided to initially create IOCBs and to indicate the action to be taken when a reply is received. In addition, commands are provided to tabulate and count data as well as to make entries in the game log. A detailed description of each of the commands is given in Appendix G; a brief summary is given here.

Message to Umpire. The "Message To Umpire" command will cause a message to be sent to the umpire. No reply will be anticipated. Any variable from the program can be inserted into the message.

Message to Player. The "Message To Player" command will send a message to any single player, any group of players and/or the umpire. No reply will be expected. A variable from the Control Program can be inserted into the text of the message.

Message to Log. The "Message To Log" command will tabulate the variable indicated in the table indicated. This information is available for offline processing after the play of the game.

Message For Reply. The "Message For Reply" command will send a message to any single player, any group of players and/or the umpire. A reply will be expected and the disposition of the reply information will be coded with this command. A variable from the Control Program can be inserted into the text of the message that is sent requesting the reply.

Tabulate. The "Tabulate" command will tabulate the variable indicated in the table indicated. This information is available for offline processing after the play of the game.

Count. The "Count" command is used to increment the indicated counter. This information is available after the play of the game for offline processing.

CHAPTER XI

USING THE SYSTEM

The Real Time Gaming System has been designed to make the steps between the decision to run a game and the actual running of the game as simple as possible. Many of the complex steps in programming a real-time system have been built into the monitor system and do not need to be considered when designing a game. The major steps in running a desired game on the system are: design of the game, programming, system generation, and running the game. Each will be discussed separately.

Design of the Game

When the decision is made to run a game under the Real Time Gaming System, it will be necessary to design the game within the framework of the system. If the game of interest already exists as an event-store game, it will probably lend itself well for adaptation to the system.

The flow of logic, decisions, information, etc., must be formulated in terms of events. It is necessary to decide what are significant events in the game, what are the alternatives to various courses of action, what information is available at different stages of the play of the game, etc. This information must be formulated into an overall plan for the game. One way of representing this information might be in a logic flowchart.

Programming

When the design of the game is firm, it is necessary to code the Control Program. This is done primarily in the RTGS Control Program Command Language supplemented by the OS/360 Assembler Language and Fortran IV. Each of the event routines is programmed as a self-contained block of coding. Any subroutines which may be required are programmed; then, any general coding needed for the game initialization is provided.

There must be at least one block of coding provided which is neither a subroutine nor an event routine. The first such block which appears in the Control Program will be considered the entry point for the start of the game and initialization.

System Generation

When all of the programming is finished, it will be run through the system generation phase. This consists of two stages: pre-processing and object system generation.

Pre-processing. During the pre-processing stage, the syntax of the source statements will be checked and diagnostics will be produced to indicate errors in the source statements. If there are any errors, they will be listed; if there are none, the source program will be edited into a new form acceptable to the RTGS system generation stage. All of the Fortran IV coding will be removed and compiled separately.

Object-system generation. If the pre-processing stage is successful, the statements, in their new form, will be

compiled and assembled into the final version. The result is a set of object modules ready to link edit into a single module ready for execution.

Running of the Game

When the game is to be run, all of the players station themselves at various remote terminals. At some pre-determined time, they dial-in using the installation's standard tele-processing procedures. The only requirement is that the umpire must be the first person to dial in. After the umpire checks into the system, the whole logging-on process will be self-coaching by the system and it is only necessary to follow instructions.

The play of the game will proceed until the first time the "end-of-game" command is encountered.

CONCLUDING REMARKS

The Real Time Gaming System, as proposed here, has not been fully implemented as of the time of the submission of this thesis. Most of the details in program logic have been worked out and enough implementation has been done to determine that the system is feasible and will run with a reasonable degree of efficiency. The author intends to continue research and implementation of this system; it is anticipated that a fully operative system will be available at the Naval Postgraduate School within a year.

BIBLIOGRAPHY

American Standards Association. FORTRAN. Communications of the ACM, Vol. VII, No. 10. Baltimore: Association for Computing Machinery, 1964.

Columbia University. Conversational Terminal Access Method (CTAM). Columbia University Computing Center Report CUCC-R-13. New York: Columbia University, 1967.

IBM Corporation. IBM System/360 Operating System Assembler Language. Form C28-6514. Poughkeepsie: IBM Corporation, 1964.

IBM Corporation. IBM System/360 Operating System Fortran IV Language. Form C28-6515. Poughkeepsie: IBM Corporation, 1966.

IBM Corporation. IBM System/360 Operating System Linkage Editor. Form C28-6538. Poughkeepsie: IBM Corporation, 1966.

IBM Corporation. IBM System/360 Operating System Supervisor and Data Management Services. Form C28-6646. Poughkeepsie: IBM Corporation, 1967.

APPENDIX A

SYSTEM CONTROL BLOCKS

Each of the system control blocks shown here are represented in bytes of core storage. The numbers down the left-hand edge of the blocks represent the displacement from the beginning of the block.

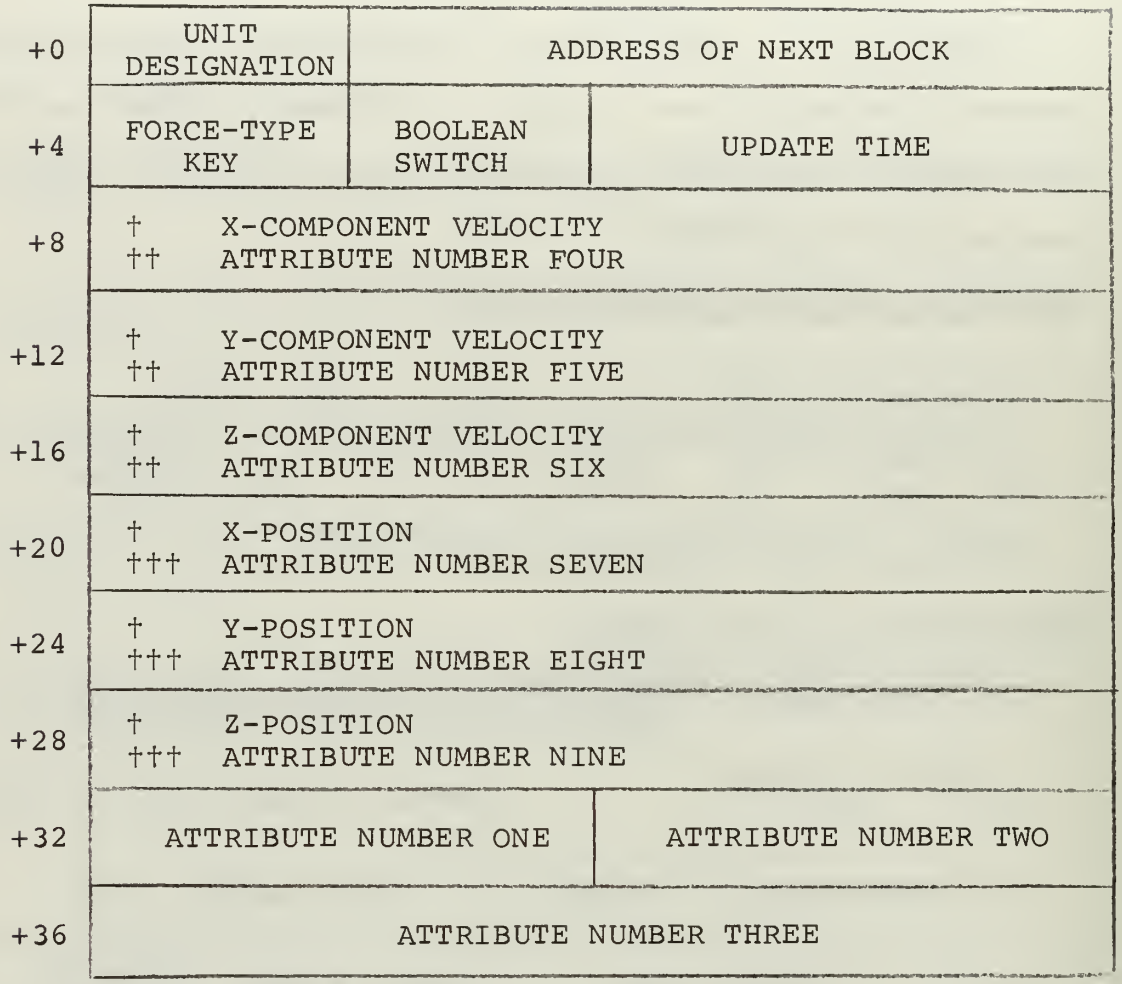
Input/Output Queue Control Block

+0	MESSAGE COUNT	REPLY COUNT
+4	QUEUE-STATUS FLAG	ADDRESS OF FIRST IOCB IN CHAIN

Input/Output Control Block

+0	REPLY FLAG	REPLY SERIAL NUMBER	LENGTH OF THIS IOCG	
+4	END-OF-QUEUE FLAG	ADDRESS OF NEXT IOCB IN CHAIN		
+8	LENGTH KEY	UNUSED	TIME MESSAGE SENT	
+12	OFFSET TIME		REPLY EVENT	BOOLEAN SWITCH
+16	ACTUAL TEXT OF THE MESSAGE ...			
+8n-4	... END OF TEXT	FILLER TO MAKE BLOCK AN EVEN MULTIPLE OF EIGHT.		

Force Status Control Block



† For continuous movement and position force

†† For discrete-motion, continuous-position force

††† For stationary forces

Event Queue Control Block

+0	KEY#1	POINTER TO TOP EVENT CONTROL BLOCK
+4	NUMBER OF EVENTS ON QUEUE	

Event Control Block

+0	KEY#1	KEY#2	POINTER TO NEXT EVENT	
+4	TIME EVENT DUE		EVENT #	BOOLEAN KEY
+8	MODIFIER NUMBER 1		MODIFIER NUMBER 2	
+12	MODIFIER NUMBER 3			

APPENDIX B

STATUS-OF-FORCES COMMANDS

Many of the operands that appear in the status-of-forces commands are common to some or all of the commands, hence, they will be described in some detail first.

Unit Designation

The unit-designation operand is the unique integer in the range (0-255) which is used for reference to the force-unit by the generated Control Program as well as by the system. When coded, the fixed-point symbolic or the fixed-point constant form of the operand may be used.

Force-Type Key

The force-type key only appears in the "New Force" command and once a force-type has been designated it may not be changed. The key consists of either one letter which stands alone or three letters which must all be present and in the correct order; they are:

1. Type designation. If the force-unit is stationary, an "S" is coded alone and no other codes are required; under this condition there are seven fullword attributes and two halfword attributes available. "Fullword" and "halfword" refer to the storage capacity of various storage locations. Table B-2 gives the range of values that can be stored in each size. If the force unit is movable an "M" is coded in the first position. For a movable force-unit the number of attributes available will vary. When the "M" is coded the following two codes are also required:

2. Force-motion key. If the motion of the force-unit is discrete, i.e., if the velocity components are meaningless, a "D" is coded in the second position. If the motion is continuous, a "C" is coded in the second position.

3. Force-positioning key. If the positioning of the forces is discrete, i.e., if the relative position is ordinal and distance between units is meaningless, a "D" is coded in the third position. If, on the other hand, the positioning of the forces is continuous, i.e., distance along the coordinate directions is a meaningful quantity, a "C" is coded in the third position.

The only combinations which may appear are listed here with the number of attributes available to be assigned to the force-unit.

FORCE-TYPE KEY	FULLWORD ATTRIBUTES	HALFWORD ATTRIBUTES
S	7	2
MDD	4	2
MDC	4	2
MCD	1	2
MCC	1	2

Table B-1

Boolean Switches

The boolean switches are available in the same form to all force-unit types. The purpose of the switches is to store decision-making information of a binary nature about the force-unit. When a force is initially defined by the "New

Force" command all switches will be defaulted to the "OFF" position unless the "boolean switch" operand is coded. If provided, it is given as an eight bit binary number indicating a 1 in the position(s) where the switch is to be turned ON. For example, if it is desired to turn on the third and fifth switches, the operand is coded as 00101000. The coding of the operand in the commands which change the individual switches is in a list form. The list is coded in parenthesis in the form (S1,S2,···); where the individual operands can be either fixed-point symbolic references or fixed-point constants. Each must be an integer in the range (1-8) representing one of the eight switches.

Time

The "time" operand, when coded, is used to indicate the game-time at which the force-unit will be referenced. It is specified in seconds since the beginning of the game, hence, it must be either a fixed-point symbolic reference or a fixed-point constant in the range (0-65535). Note: it is possible to make references in the past and future; however, the present velocities and attributes will be used to compute what the status was or will be at the reference time, hence it is possible to get a false status of the force-unit if changes have occurred or will occur between the present time and the reference time. Since references in the past represent history, it is never possible to make a retroactive change; similarly, a change in the future cannot be made ahead of time. On all references that involve change, the time parameter should be left out. If it is desired to ensure that

the change occurs at an exact time, the time can be specified and corrections for delay of the system from the real-time will be made to record the change at the specified time. It should generally be sufficient to allow the system to assign the present game-time.

Coordinate Velocities

On force-units which are movable, the X, Y and Z coordinate velocities are provided in their respective operands in order to describe the motion of the force-unit in the theater of operations. When initially defining new forces, the velocities are defaulted to zero unless they are specified. The operands may be either fixed or floating-point and may be either symbolic or constant. When making changes, it is only necessary to provide the operands for the coordinate directions that are being changed. The velocity is given in distance units per second along the axis directions.

Coordinate Positions

For movable forces which are positioned discretely, the operands are either fixed-point symbolic references or fixed-point constants placing the unit on the playing grid in its relative ordinal position. For force-units which are positioned continuously, the operands are either fixed or floating-point and either symbolic references or constants which describe the distance from the origin along the three coordinate directions. As in velocities, only those distances being set or changed need be coded.

Attributes

The number of attributes which are available varies with the force-unit type. The number that are allowed under the various types is tabulated in Table B-1. The fullword attributes may be fixed or floating-point; the halfword attributes may only be fixed-point. The range of the various attributes is given in Table B-2.

WORD SIZE	WORD TYPE	RANGE
Fullword	Floating-point	$\pm 7 \times 10^{75}$
Fullword	Fixed-point	$\pm 2.1 \times 10^{9}$
Halfword	Fixed-point	$\pm$ 32,767

Table B-2

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<NEW FORCE> / NF}	u,{MCD / MCC} [,b,t,vx,vy,vz,px,py,pz, a1,a2,a3]
[name]	{<NEW FORCE> / NF}	u,{MDD / MDC} [,b,t,px,py,pz,a1,···,a6]
[name]	{<NEW FORCE> / NF}	u,S[,b,t,a1,···,a9]

WHERE

u = unit designation; integer in range (0-255); fixed-point symbolic or constant.

b = boolean switch; binary number with eight bits.

t = reference time; integer in range (0-65535), fixed-point symbolic or constant.

vx = x-direction velocity component; fixed or floating-point fullword, symbolic or constant. Similarly for vy and vz.

px = position along x-axis; fixed-point symbolic or constant for discrete and fixed or floating-point symbolic or constant for continuous. Similarly for py and pz.

a1 = attribute #1; fixed or floating-point symbolic or constant for fullwords and fixed-point only for halfwords. Similarly for a2,a3,···a9.

Function

The function of this command is to introduce a new force-unit into the theater of operations. If a force-unit with this designation already exists it will be deleted. Once the force-unit type has been designated by this command it may not be changed during the play of the game.

Example

1. In this example it is desired to define an aircraft initially at a distance of ten nautical miles from the grid origin, on the x-axis, and at an altitude of six thousand feet. The initial velocity is inbound toward the grid origin at two hundred knots. This aircraft is to be given the force-unit designation twenty-six, attribute number one is to have a value of ten, and the boolean switch number two is to be set ON indicating "enemy" aircraft.

The command to define this new force-unit would be:

<NEW FORCE> 26,MCC,01000000,,#-277.7,,,20000,,2000,10 .

The floating-point -277.7 represents the velocity in yards per second. The fixed-point 20000 represents the distance in yards. Note that only those factors which are non-zero need be included. In this case the present game-time would be assigned to this position.

2. Place a King (represented by boolean switch number one) on the fourth row and third column of a chess board. Set attribute number one to a value of one to indicate that this is the black King. Designate as force-unit number one.

The command to define this new force-unit would be:

NF 1,MDD,10000000,3,4,,1 .

Note that in this case the short form of the command was used.

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<DELETE FORCE> / DF}	u[,p1,p2,···,pn]

WHERE

u = unit designation; integer in range (0-255); fixed-point symbolic or constant.

p1 = player to be notified; integer in range (0-11); fixed-point symbolic or constant. Similarly for p2,p3,···,pn.

Function

The function of this command is to delete the indicated force-unit from the theater of operations. The unit designation indicated is now free for use by some other force-unit. The players listed in the "p" operands will be given a message to indicate that the unit was deleted. The message will appear in the form:

0246:35 FORCE-UNIT 6 DELETED .

Example

1. It is desired to cancel force-unit six and notify the umpire as well as player number one that the cancellation has occurred. The following command will accomplish this:

<DELETE FORCE> 6,0,1 .

2. Assume that it is not desired that any players be notified of the cancellation and that a fixed-point value of six is already stored in location "SIX"; then the command could be given as:

<DELETE FORCE> SIX .

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<READ FORCE> / RF}	u[,v1,v2,···,v11]

WHERE

u = unit designation; integer in range (0-255); fixed-point symbolic or constant.

v1,v2,··· = list of variable symbolic names which are to receive the status information currently stored for the force-unit. The position of the variables in the list is the same as that in the <NEW FORCE> command used to define the force-unit, starting with the boolean switch operand.

Function

The function of this command is to update the FSCB for the designated unit and return the requested information. This is done by placing the information into the locations indicated by the symbolic references which appear in the operand list. Only those factors which are needed should be requested.

Example

1. Suppose we desire to know the distance of force-unit twenty-six, which was the aircraft defined in the example given under <NEW FORCE>, along the x-axis. We desire to place this value into the location symbolically represented by the name POSIT; the following command could be given:

<READ FORCE> 26,,,POSIT .

Command Description

NAME	COMMAND	OPERAND(S)
[name]	{<CHANGE FORCE> / CF}	u[,b,t,vx,vy,vz,px,py,pz, a1,a2,a3]
[name]	{<CHANGE FORCE> / CF}	u[,b,t,px,py,pz,a1,···,a6]
[name]	{<CHANGE FORCE> / CF}	u[,b,t,a1,···,a9]

WHERE

u = unit designation; integer in range (0-255); fixed-point symbolic or constant.

b = boolean switch; binary number with eight bits.

t = reference time; integer in range (0-65535); fixed-point symbolic or constant.

vx = x-direction velocity component; fixed or floating-point fullword, symbolic or constant. Similarly for vy and vz.

px = position along x-axis; fixed-point symbolic or constant for discrete and fixed or floating-point symbolic or constant for continuous. Similarly for py and pz.

a1 = attribute#1; fixed or floating-point symbolic or constant for fullword and fixed-point only for halfwords. Similarly for a2,a3,···,a9.

NOTE: The first form is used for MCD or MCC type force-units; the second form for MDD or MDC type force-units; and the last for S type force-units.

Function

The function of this command is to change any, or all, of the characteristics of a force-unit. If the force-unit has not been defined the command will take no effect. Only those characteristics which are to be changed are listed.

The others are left out and the corresponding characteristics will remain unchanged. If the "time" operand is not coded, the change will occur at the present game-time. No change can be made in the past or the future since it may effect the play of the game in the intervening time. It is possible, however, to designate a time to ensure that the change is recorded at the correct time. If a time is coded, it must not be too far off the present game-time and specifying it as such can cause unpredictable results.

Example

Assume that a force-unit designated as six is of the MDD type and it is desired to set attribute number two to a fixed-point value of seventeen. In addition, attribute number three is to be set equal to the floating-point value currently at the location with the symbolic reference #D1. The following command would be used:

<CHANGE FORCE> 6,,,,,,17,#D1 .

If, on the other hand, this same command were issued and force-unit six was stationary (type "S") then the result would be that attribute number four would be set to a value of seventeen and attribute number five would be set equal to the floating-point value currently at location #D1.

Command description

NAME	COMMAND	OPERANDS
[name]	{<DISTANCE BETWEEN>} {DB}	ul,u2,d ,t,({1 0},{1 0},{1 0})

WHERE

ul = unit designation; integer in range (0-255); fixed-point symbolic or constant. Similarly for u2.

d = distance; fixed or floating-point symbolic reference.

t = reference time; integer in range (0-65535); fixed-point symbolic.

(x,y,z) = component reference list; integers in the range (0,1); fixed-point symbolic or constant. Defaults to (0,0,0).

Function

The function of this command is to compute the true distance between two units in the theater of operations and place the value in the variable named by the "d" operand. If the "t" operand is coded, the reference time will be provided. The list (x,y,z) is coded with a 1 to indicate the coordinates in which the distances are to be measured and 0 in the coordinates which are to be skipped. If the list is coded (1,1,1) slant range will be given; whereas if (1,1,0) is coded, horizontal range will be given. If the list is not coded, (1,1,1) will be assumed.

Example

1. Suppose we desire horizontal range from force-unit six to force-unit seven. We want the result to be truncated to the nearest integer, converted to fixed-point and stored in

the location with the symbolic reference "DIST". The following command will be used:

```
<DISTANCE BETWEEN>  6,7,DIST,,(1,1,0)  .
```

2. Suppose now that we desire the altitude of the aircraft with force-unit designation nine and that the nine has been stored in a location designated ACNO. Suppose also that we want the computed value to be placed in the location symbolically referenced as DISTAC and we want the time of the reference to be placed into location TME. Then we want to read the position of all of the boolean switches for this same aircraft into location WHICH. We are force-unit six and the altitude measurement is to be made relative to our altitude. The following sequence of commands could be used:

```
BRP4  <DISTANCE BETWEEN>  6,ACNO,DISTAC,TME,(0,0,1)
      <READ FORCE>        ACNO,WHICH,TME          .
```

APPENDIX C

EVENT MANAGEMENT COMMANDS

Many of the operands that appear in the event management commands are common to some or all of the commands; hence, they will be described in some detail first.

Event Designation

The event-designation operand is an integer in the range (0-255) which is used to reference the event routine by the individual events. When coded, the fixed-point symbolic or the fixed-point constant form of the operand may be used.

Boolean Switches

The boolean switches are available in the same form to all event routines. The purpose of the switches is to store decision-making information of a binary nature. When an event is stored by the "Store Event" command all switches will be defaulted to the "OFF" position unless the "boolean switch" operand is coded. If provided, it is given as an eight bit binary number indicating a 1 in the position(s) where the switch is to be turned "ON". For example, if it is desired to turn "ON" the third and fifth switches, the operand is coded 00101000. The coding of the operand in the commands which change the individual switches is in a list form. The list is coded in parenthesis in the form (S1,S2,···), where the individual operands can be either fixed-point symbolic references or fixed-point constants. Each must be an integer in the range (1-8) representing one of the eight switches. A full set of switches is associated

with each event stored in the queue; they may be referenced by the event routine during execution by the symbolic references: $ES1,$ES2,···,$ES8.

Time

The "time" operand is used to indicate the game-time at which the event is to occur. It is specified in seconds since the beginning of the game, hence, it must be either a fixed-point symbolic reference or a fixed-point constant in the range (0-65535). When an event is stored with a time which has already passed, it will be put at the top of the event queue and flagged for immediate execution at the first opportunity.

Modifiers

Three modifiers are available to each event in order to pass information to the event routine at execution time. Modifiers number one and two are halfwords, hence, they must be coded as fixed-point symbolic references or as fixed-point constants. Modifier number three, on the other hand, is a fullword and may be coded as fixed or floating-point and either symbolic or constant. The range of the modifiers is given in Table B-2 in Appendix B. Within the event routine the modifiers may be referred to symbolically by the references: $M1, $M2, and $M3.

Command description

NAME	COMMAND	OPERAND
none	{<DEFINE EVENT ROUTINE> / DER}	e
WHERE		
e = event designation; integer in range (0-255); fixed-point symbolic or constant		

Function

This command functions as a header to identify the beginning of an event routine. The commands that follow this one in the Control Program constitute the sequence of commands that are to be executed when the designated event comes due. Note: This command may not contain a reference name; if one is supplied it will be ignored.

Example

Assume that we wish to define the sequence of events constituting event three. The following coding would be supplied:

```
<DEFINE EVENT ROUTINE>  3
 first command...
 second command...
      etc.
```

Command description

NAME	COMMAND	OPERAND
[name]	{<END EVENT ROUTINE> / EER}	none

Function

This command functions as the last command in an event routine. It marks the end of the coding for the event routine and must be present. The next command in the Control Program must have a symbolic reference in the name field since control is never passed to the next command after this one.

Example

Continuing the example given under <DEFINE EVENT ROUTINE>, the <END EVENT ROUTINE> command would be inserted at the end of the block of coding. The resulting coding would constitute a complete event routine.

```
<DEFINE EVENT ROUTINE>  3
 first command...
 second command...

 last command...
<END EVENT ROUTINE>
```

Command description

NAME	COMMAND	OPERAND
[name]	{<EVENT EXIT> / EXIT}	none

Function

The function of this command is to provide for multiple exits from an event routine. When this command is encountered in the event routine coding, an exit is taken and control is returned to the Control Program as though the entire event routine were executed. It is not necessary to place this command in front of the <END EVENT ROUTINE> command.

Example

Assuming that it is desired to take and exit from the event routine that was defined in the two previous command examples, we might have done so with the following sequence:

```
          <DEFINE EVENT ROUTINE>   3
           first command...
           second command...
          <EVENT EXIT>
     BRP   command...

           last command...
          <END EVENT ROUTINE>
```

Note: In this example, when the <EVENT EXIT> command was used, it was necessary to give a symbolic reference name to the next command in the sequence.

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<STORE EVENT> / STE}	e,t[,b,m1,m2,m3]

WHERE

e = event designation; integer in range (0-255); fixed-point symbolic or constant.

t = execution time; integer in range (0-65535); fixed-point symbolic or constant.

b = boolean switch; binary number with eight bits.

m1 = modifier number one; halfword; fixed-point symbolic or constant. Similarly for m2.

m3 = modifier number three; fullword; fixed or floating-point symbolic reference or constant.

Function

The function of this command is to cause one event to be placed in the event queue for execution at the game-time indicated in the "t" operand. The event routine that corresponds with the event designation must have been defined by a <DEFINE EVENT ROUTINE> command. The <STORE EVENT> command may appear anywhere in the Control Program coding including event routines; it may also appear inside the event routine which it is calling.

Example

Suppose we want to cause the event routine designated as event six to be executed at time 1776 and modifier two is to have a fixed-point value of nineteen. We could code:

```
<STORE EVENT>   6,1776,,,19
```

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<EXECUTE EVENT> / EXE}	e[,b,m1,m2,m3]
WHERE		
All operands are exactly as coded in the <STORE EVENT> command. The "t" operand is not present.		

Function

The purpose of this command is to cause the immediate execution of the designated event. When the event routine is completed, control is returned to the next command in the Control Program. Note: The <EXECUTE EVENT> command may not be issued within the event routine which it is calling.

Example

1. Suppose that in the Control Program coding it is necessary to execute event routine number six with modifier number three set to a floating-point seventeen. The following coding could be used:

```
EXE   6,,,,#17
```

2. Suppose now that among other things, event routine six must execute event routine seven, and that it must do so by passing modifier number three on to event routine seven.

```
<DEFINE EVENT ROUTINE>   6
 command...
<EXECUTE EVENT>  7,,,,$M3
```

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<CANCEL EVENT> / CE}	[e,t,b]
WHERE		
e,t, and b are as defined in <STORE EVENT> command. While all three operands are optional, at least one of them must be coded.		

Function

This command will find the first event in the queue which exactly matches the operands that are coded; this event will be canceled. Only those operands which are supplied will be used in the test. If more than one event in the queue has the characteristics given, only the first will be canceled. If no event is found with the characteristics, no action will result.

Example

Suppose that we had previously stored an event for event routine three to be executed at time 3778; and we now desire to cancel it. The following command would be used:

```
<CANCEL EVENT>   3,3778
```

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<GLOBAL EVENT CANCEL> / GEC}	[e,t,b]
WHERE		
e,t, and b are as defined in <STORE EVENT> command. While all three operands are optional, at least one of them must be supplied.		

Function

This command will find all of the events in the queue which exactly match the operands that are coded; these will all be canceled. Only those operands which are supplied will be used in the test. If no events which match are found, no action will result.

Example

1. Suppose it is desired that any event number three which is currently in the queue be canceled. The following command will cause this to be done:

<GLOBAL EVENT CANCEL> 3

2. If it is desired that all events with the eighth boolean switch "ON" and all others "OFF" be canceled, the following command could be used:

GEC ,,00000001

Note that in this case the short form of the command was used.

APPENDIX D

LINKAGE COMMANDS

The linkage commands are provided in order to give a means of transferring control from one section of coding to another. In all cases, the standard conventions for subroutines for the OS/ 360 will be used. When a parameter list is passed to a subroutine or is defined in the command "Subroutine", it will appear in the list form which consists of a list of variable references enclosed in parenthesis. When Fortran IV coding is mixed with the RTGS Command Language, it is only necessary to list the symbolic references as operands in an open-ended operand list.

COMMANDS

Command description

NAME	COMMAND	OPERAND
[name]	{ <END OF GAME> / END }	none

Function

The "End of Game" command is issued when the game is to be stopped for the last time. This command will cause all of the data collection, etc., to be performed. Any events which are still due for execution will be tabulated. The running log of the game will be brought up to date. When all of the end-of-game functions have been completed, control will be transferred to the computer's operating system to terminate the run.

Command description

NAME	COMMAND	OPERAND
[name]	{<ASSEMBLY LANGUAGE> / AL}	none
[name]	{<END ASSEMBLY LANGUAGE> / EAL}	none

Function

The function of the "Assembly Language" command is to indicate that the coding which follows in the Control Program will be O/S 360 Assembler Language. All general purpose and floating-point registers are free for use in the coding which follows. The "End Assembly Language" command serves to indicate that the last Assembler Language source statement has been coded and that programming will be in RTGS Command Language again.

Example

A typical sequence which might be used to clear the first seven boolean switches to "OFF" and return them to their force-unit might be:

```
<READ FORCE>  6,$T1
<ASSEMBLY LANGUAGE>
 NI  $T1,X'1'
<END ASSEMBLY LANGUAGE>
<CHANGE FORCE>  6,$T1
```

The "Read Force" command brings out the current value of force-unit six's boolean switches; this word is placed into symbolic location "$T1", then the "And" operation is done to change the first seven bits to zero. The adjusted word is then inserted back into the FSCB.

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<FORTRAN> / FTN}	[parm1,parm2,···]
[name]	{<END FORTRAN> / EFTN}	none

WHERE

parm1 = first symbolic reference; fixed or floating-point symbolic reference. Similarly for parm2,···

Function

The function of the "Fortran" command is to indicate that the coding which follows in the Control Program will be Fortran IV. It is necessary to place in the operand list, in any order, those symbolic references which will be carried over from the RTGS Command Language coding to the Fortran source coding. Those symbolic references used in the Fortran IV source coding which are not in the operand list will be considered independent for this block of Fortran IV coding. The "End Fortran" command serves the same purpose as the "End Assembly Language" command.

Example

```
<READ FORCE>  6,,,#$T1,#$T2,#$T3
<FORTRAN>     #$T1,#$T2
  A = #$T1 + #$T2 + 6.3
  #$T2 = A / 7.0
<END FORTRAN>
```

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<SUBROUTINE> / SUB}	[(parmlist)]
[name]	{<END SUBROUTINE> / ESUB}	none
[name]	{<RETURN> / RET}	none

WHERE

parmlist = list of symbolic references; fixed or floating-point; enclosed in parenthesis.

Function

The function of the "Subroutine" command is to indicate that the RTGS Command Language source coding which follows will constitute a subroutine. The parameter list will be passed to the subroutine using standard conventions for linkage. The "End Subroutine" command marks the last statement in the subroutine and must be coded to separate the block of coding from that which follows. The "Return" command allows multiple exits to be taken from the subroutine at any point. Any number of "Return" commands may appear within a subroutine. The subroutine must be self-contained and not have external linkages other than by the entry point and "return" commands. The "End Subroutine" command performs all of the functions of the "Return".

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<EXECUTE SUBROUTINE> / CALL}	nl[l,(parmlist)]
[name]	{<RETURN CODE BRANCH> / RCB}	brp1,brp2[,brp3,···]

WHERE

nl = subroutine name; symbolic name reference.

parmlist = parameter list for subroutine, fixed or floating-point symbolic references or constants which match the subroutine parameter list.

brp1 = branch point name; symbolic name reference to be taken when return code is zero.

brp2 = branch point name; symbolic name reference to be taken when return code is four. Similarly for brp3,···

Function

The function of the "Execute Subroutine" command is to cause a branch to the designated subroutine with the parameter list as specified. The Control Program will transfer control to the subroutine for execution and will pass the parameter list in standard format. When the subroutine has completed execution, control will be transferred to the next command in the Control Program. If standard register fifteen return code linkage conventions have been used, the "Return Code Branch" can be used to cause multiple branching to the designated branch points.

Command description

NAME	COMMAND	OPERANDS
[name]	{<RANDOM VARIABLE> / RV}	gen,v

WHERE

gen = generator designation; symbolic name reference.

v = variable to receive random variable; floating-point symbolic reference.

Function

The function of this command is to draw the next random variable from the generator specified in the "gen" operand. The variable will be placed in the location specified by the "v" operand. At system generation time the probability distribution from which the random variable will be drawn is specified as "Uniform", "Poisson (Exponential)", or "Normal". Any number of generators can be specified in order to maintain independence.

Example

Suppose that random number generator three had been designated as "Normal" with mean four and standard deviation one. The next random variable from this distribution may be drawn by issuing the following command:

<RANDOM VARIABLE> 3,#RV

The value returned would be anywhere in the range $(-\infty,+\infty)$; however, it would most likely be in the range (+1.0, +7.0).

Command description

NAME	COMMAND	OPERAND
[name]	{<RETURN TO MONITOR> / RTM}	none

Function

The "Return to Monitor" command is provided as the standard end to a logical block of coding. For example, the coding which is influenced by the "Do Periodically" command must eventually return control to some other segment of the Control Program or to the monitor system. When the command is encountered in the coding stream, control passes from the Control Program back to the system monitor. The next command in the coding must have a name in the name field in order to provide a means for it to be executed.

Example

Suppose that it is desired to execute event routine number six every thirty seconds. The following block of coding could be used:

```
        command...
        <DO PERIODICALLY> ,30
        <EXECUTE EVENT>   6
        <RETURN TO MONITOR>
BRP     command...
```

APPENDIX E

TIMING COMMANDS

Most of the operands that appear in the timing commands are common to several of the commands; hence, they will be described in some detail first.

Time

The "time" operand, when coded, is used to indicate either game-time or clock-time. In the case of game-time it is specified in seconds since the beginning of the game; hence, it must be either a fixed-point symbolic reference or a fixed-point constant in the range (0,65535). In the case of clock-time, it is game-time plus some base time corrected to hours, minutes, and seconds; hence, it is represented as a set of three fixed-point symbolic references or fixed-point constants or combination.

Branch Points

Branch points are coded in order to symbolically represent the entry point for control at specified times. They are specified as symbolic name references.

Time Increment

The "Do Periodically" command requires a time increment to be coded. It is represented in seconds and must be specified as a fixed-point symbolic reference or a fixed-point constant.

Command description

NAME	COMMAND	OPERAND
[name]	{<SET TIME> / TIME}	tl
WHERE		
tl = game-time setting; integer in range (0,65535); fixed-point symbolic or constant.		

Function

The function of the "Set Time" command is to set the master clock to the time indicated in the "tl" operand. As a result of this command the master clock will be stopped and reset with the time specified. The clock will then remain stopped until started with another command.

Example

1. The normal use of this command would be during the game initialization phase. The following command could be given to set the clock up for a run:

<SET TIME> 0

2. Suppose that it is necessary to go back and pick up the game at time one hour. The following command could be given:

<SET TIME> 3600

It should be noted, however, that this would not necessarily duplicate the activities of an earlier run.

Command description

NAME	COMMAND	OPERAND
[name]	{<START TIME> / GO}	none
[name]	{<STOP TIME> / STOP}	none

Function

The function of the "Start Time" command is to start the master clock; the setting presently on the master clock will be retained. The "Stop Time" command will cause the master clock to stop; the setting will not be altered as a result of this command.

Example

During game initialization, the following sequence of commands could be used to get started:

```
<SET TIME>    0
<START TIME>
 command...
```

Warning

Care must be exercised when using this command. If the clock has been stopped by a "stop time" command, it must be restarted again by some coding which is able to be reached by the Control Program during the present time frame, i.e., if the "Start Time" command is located in an event routine which is not due for execution, the command will never be reached since the events are frozen by the stopped time.

Command description

NAME	COMMAND	OPERAND
[name]	{<SET CLOCK> / SC}	tl,(h,m,s)

WHERE

tl = game-time; integer in range (0,65535); fixed-point symbolic or constant.

h = clock-time hours component; integer in the range (0,23); fixed-point symbolic or constant.

m = clock-time minutes component; integer in the range (0,59); fixed-point symbolic or constant.

s = clock-time seconds component; integer in the range (0,59); fixed-point symbolic or constant.

Function

The function of the "Set Clock" command is to establish the relationship between game-time and clock-time. The "tl" operand is coded with a specific game-time and the "(h,m,s)" operand is coded with the corresponding clock-time. The system will use this information to set the base time which will be used in the calculation of clock-times.

Example

Suppose that when the game is started the clock-time is supposed to be 1330:00; the following sequence of commands could be used in the initialization phase:

```
<SET TIME>    0
<SET CLOCK>   0,(13,30,00)
<START TIME>
 command...
```

Command description

NAME	COMMAND	OPERAND(S)
[name]	{<PRESENT TIME> / PT}	t
[name]	{<PRESENT CLOCK> / PC}	[h,m,s]

WHERE

t = game-time; fixed-point symbolic reference.

h = hour component of clock-time; fixed-point symbolic reference.

m = minute component of clock-time; fixed-point symbolic reference.

s = seconds component of clock-time; fixed-point symbolic reference.

Function

The function of the "Present Time" command is to return the present value of the game-time to the variable designated in the "t" operand. The "Present Clock" command will return the components of clock-time which are coded to the locations specified by the symbolic references.

Example

Suppose during the play of the game it is necessary to know the game-time and clock-time. The following sequence of commands could be issued:

```
<PRESENT TIME>    $T1
<PRESENT CLOCK>   $T2,$T3,$T4
```

As a result of these commands the required values will be available in the first four temporary storage locations.

Command description

NAME	COMMAND	OPERANDS
[name]	{<SIGNAL AT TIME> / SAT}	t2,brpl
[name]	{<SIGNAL AFTER LAPSE> / SAL}	t1,brpl

WHERE

t1 = lapse time; integer in range (0,65535); fixed-point symbolic or constant.

t2 = game-time; integer in range (0,65535); fixed-point symbolic or constant.

brpl = symbolic reference to a branch point.

Function

The function of the "Signal at Time" command is to indicate to the system monitor that control is to be started at the point in the Control Program referenced by the second operand, at the time indicated by the first operand. The "Signal After Lapse" command causes the same thing to occur with the exception that the first operand is a time lapse between present game-time and the game-time that the branch is to occur.

Example

Assume that there is a block of coding which has a symbolic branch point name "BRP" and that it is desired that this coding be executed at game-time 3675. The following command could be given during initialization:

```
<SIGNAL AT TIME>   3675,BRP
```

Command description

NAME	COMMAND	OPERAND
[name]	{<DELAY UNTIL> / DU}	t1
[name]	{<DELAY FOR> / DF}	t2
WHERE		
t1 = game-time; integer in range (0,65535); fixed-point symbolic or constant.		
t2 = delay time; integer in range (0,65535); fixed-point symbolic or constant.		

Function

The function of the "Delay Until" command is to cause the execution of the commands in the Control Program sequence to cease until the time designated in the operand. In the case of the "Delay For" command, the same thing occurs; however, the operand is interpreted as a delay rather than an actual time. Meanwhile, the Control Program will execute any unexecuted event routines or do any other work which is backlogged. When the time indicated comes due, the Control Program will continue executing the next command in the sequence.

Warning

When using this command, care should be exercised to keep track of the value of any variables in the Control Program. There is nothing to prevent the value of a variable which has been set prior to the delay from being changed by some other block of coding which is executing during the delay.

Command description

NAME	COMMAND	OPERANDS
[name]	{<DO PERIODICALLY>} DO	[st,inc,ni,ct]

WHERE

st = start time; integer in range (0,65535); fixed-point symbolic or constant; defaults to the present game-time.

inc = time increment; integer in range (0,65535); fixed-point symbolic or constant; defaults to 60 seconds.

ni = number of iterations; integer in range (0,65535); fixed-point symbolic or constant; defaults to 1000.

ct = completion time; integer in range (0,65535); fixed-point symbolic or constant; defaults to end of game (65535).

Function

The function of the "Do Periodically" command is to set up the monitor system to automatically branch to the sequence of commands which follow this one repeatedly with a cycle determined by the "inc" operand. Beginning at the starting time indicated in the "st" operand, the sequence will be executed every "inc" seconds until one of two conditions occurs:

1. The number of iterations specified in the "ni" operand is met.

2. The completion time specified in the "ct" operand is encountered.

No execution of the sequence of commands will occur if any of the following conditions is true:

1. The start time is later than the completion time.

2. The completion time is later than current game-time.

3. The number of iterations is equal to zero.

The number of iterations and completion times were defaulted high in order to allow coding only that one of interest without worrying about the other one interfering. In any event both conditions are checked and whichever occurs first will govern.

This command may be nested as deeply as desired. It is possible to have an outer loop executing every hour with a nested loop executing every ten minutes for the first thirty minutes of the hour.

Warning

Over-usage of this command can cause the system to become bookkeeping bound, expecially in the case where the increment time is very short.

Example

Consider the case discussed above with nesting. The following sequence of commands could be used:

```
<DO PERIODICALLY>   ,3600,5
<DO PERIODICALLY>   ,600,4
 command...
 command...

 command...
<RETURN TO MONITOR>
```

Command description

NAME	COMMAND	OPERAND
[name]	{<CEASE> / C}	ref
WHERE		
ref = reference to statement; branch point symbolic reference.		

Function

The function of the "Cease" command is to stop the execution of a previously defined "Do Periodically" command. This command, when executed, will cause all future executions of the statements coming under the "Do Periodically" command to be canceled. The operand references the name field of the "Do Periodically" command that is being canceled.

APPENDIX F

LOGIC AND ARITHMETIC COMMANDS

Many of the operands that appear in the logic and arithmetic commands are common to some or all of the commands; hence, they will be described in some detail first.

Switch Type Designation

When reference is made to one or several boolean switches, it is necessary to indicate whether the switches are those associated with an event, a force-unit, or the general switches available at any point in the Control Program. The "switch-type" operand is coded with the letter E to indicate an event switch; this reference may only be made within the event routine of the event being tested. If the "switch-type" is coded G, the reference is to the general switches; and, if a number in the range (0,255) is coded, the reference will be assumed to be a force-unit.

Branch Points

Branch point references are made when it is necessary to designate the entry point to be used for a branch. The branch point is coded as a symbolic name reference.

Command description

NAME	COMMAND	OPERANDS
[name]	{<SET SWITCH ON>} {SWON}	{G} {u} ,(sw1,sw2,•••)
[name]	{<SET SWITCH OFF>} {SWOF}	{G} {u} ,(sw1,sw2,•••)
[name]	{<CHANGE SWITCH>} {SWCH}	{G} {u} ,(sw1,sw2,•••)

WHERE

G = general switch; coded as shown

u = force-unit designation; integer in range (0,255); fixed-point symbolic or constant.

sw1 = switch designation; integer in range (1,16); fixed-point symbolic or constant. Similarly for sw2,sw3,•••

Function

The function of the "Set Switch On" command is to turn the designated switches to the "ON" position. The operand which lists the switches is coded in the list form,i.e., enclosed in parenthesis is a list of those switches which are to be affected by the command. Notice that it is not possible to change the event switches once they have been stored in the event queue.

Example

The following command could be used to turn the third and fifth switch "ON" for force-unit seventeen:

<SET SWITCH ON> 17,(3,5)

Function

The function of the "Set Switch Off" command is, in principle, the same as the "Set Switch On" command. It will not be discussed separately.

Function

The function of the "Change Switch" command is to change the present setting of the designated switch or list of switches to their complementary position.

Example

1. Assuming that force-unit seventeen has switches number three and five "ON", either of the two commands given here could be used to turn switch five "OFF".

<CHANGE SWITCH> 17,(5)

<SET SWITCH OFF> 17,(5)

2. If it is desired to turn switch five "OFF" and at the same time turn switch seven "ON" the following command would be used.

<CHANGE SWITCH> 17,(5,7)

Command description

NAME	COMMAND	OPERAND
[name]	{<NO OPERATION> / NOP}	none
[name]	{<BRANCH> / B}	brp
WHERE		
brp = branch point; symbolic reference to name.		

Function

The function of the "No Operation" command is to act as a place to anchor a symbolic name reference for branching purposes. It may be necessary to have some other section of the Control Program branch to this section of coding; however, it may not be possible to put a symbolic name on the desired command either because it already has another name or because it is not one of the commands which allows a name field. In this case the "No Operation" command can be placed in front of this command with the name. The "No Operation" command can also serve as the temporary replacement for a block of coding which is to be added later.

The function of the "Branch" command is to cause an unconditional branch to occur to some other point in the Control Program. The entry point is coded as a symbolic name reference in the operand.

Command description

NAME	COMMAND	OPERANDS
[name]	{<BRANCH ALL ON> / ALLN}	{E / G / u} ,(s1,s2,···),brp
[name]	{<BRANCH ANY ON> / ANYN}	same as above
[name]	{<BRANCH ALL OFF> / ALLF}	same as above
[name]	{<BRANCH ANY OFF> / ANYF}	same as above

WHERE

E = event switch; coded as shown.

G = general switch; coded as shown.

u = force-unit designation; integer in range (0,255); fixed-point symbolic or constant.

s1 = switch designation; integer in range (1,16); fixed-point symbolic or constant. Similarly for s2,s3,···

brp = branch point symbolic reference

Function

The function of the "Branch All On" et al. commands is to consider the list of switches given in the second operand. If the condition called for exists a branch will occur to the branch point name symbolically represented in the third operand. When the first operand is coded "E" the command must appear within an event routine; the switch list then refers to the boolean switches in the event which caused the execution of the event routine. If the first operand is coded "G", the command can appear anywhere and the set of boolean

switches which is provided for the Control Program to use throughout the game will be used to determine the conditions for the branch. If the first operand is coded with an integer in the range (0,255), it will be interpreted to indicate one of the force-units which are presently being maintained by the status-of-forces monitor.

In any of the following conditions, the command will take no effect:

1. If the first operand is coded with an integer, indicating a force-unit, and the force-unit is not presently being kept by the status-of-forces monitor.

2. If the first operand is coded "E" and the command does not appear within an event routine.

3. If a switch designated in the switch list is outside of the range that is associated with the first operand.

Example

Suppose we desire to branch to the section of coding named BRP3 if and only if the boolean switches number one through seven in force-unit number six are in the "ON" position. We could use the following command:

```
<BRANCH ALL ON>  6,(1,2,3,4,5,6,7),BRP3
```

It would also have been possible to use:

```
<BRANCH ANY OFF>  6,(1,2,3,4,5,6,7),BRP2
<BRANCH>          BRP3
BRP2 command....
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<BRANCH ON VALUE> / BOV}	v[,brp1,brp2,brp3]

WHERE

v = test value; fixed or floating-point symbolic reference.

brp1 = branch point for minus; symbolic reference to a branch point.

brp2 = branch point for zero; symbolic reference to a branch point.

brp3 = branch point for plus; symbolic reference to a branch point.

Function

The function of the "Branch on Value" command is to consider the value of the variable named in the first operand and based on its arithmetic value make a branch. If the value of the variable is negative, the branch will be to the second operand; if zero, the branch will be to the third operand; and, if positive, the branch will be to the fourth operand. If any of the branch point operands are not coded and the corresponding condition exists, the next command in the sequence will be executed.

Command description

NAME	COMMAND	OPERANDS
[name]	{<ADD> / ADD}	res,a1,a2[,a3,···]
WHERE		
res = result; fixed or floating-point symbolic reference. a1 = addend; fixed or floating-point symbolic or constant. Similarly a2,a3,···		

Function

The function of the "Add" command is to add a list of addends and place the result into a location designated by the first operand. The list of addends may be of any length and if the symbolic reference designating the location of the result also appears in the list, the old value will be used for the calculation. Full mixed-mode (fixed-point and floating-point) arithmetic is allowed.

Example

1. The following command will increment the value of COUNT by one.

```
<ADD>  COUNT,COUNT,1
```

2. The following command will add the values of #VAL and TEST and place the result in fixed-point into the location symbolically represented as ANSWER

```
<ADD>  ANSWER,#VAL,TEST
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<SUBTRACT> / SUB}	res,min,sub

WHERE

res = result; fixed or floating-point symbolic reference.

min = minuend; fixed or floating-point symbolic reference or constant.

sub = subtrahend; fixed or floating-point symbolic reference or constant.

Function

The function of the "Subtract" command is to perform a subtraction and place the result in the location symbolically represented by the first operand. If the "result" symbolic location also appears as one of the factors, the old value will be used in the calculation. Full mixed-mode arithmetic is allowed.

Example

1. Suppose that it is desired to decrement the value of the variable located by the symbolic reference COUNT by one. The following command could be used.

```
<SUBTRACT>  COUNT,COUNT,1
```

2. The following command would do the same but the result would be converted to floating-point.

```
<SUBTRACT>  #COUNT,COUNT,1
```

NOTE: The old value of COUNT would remain unchanged since #COUNT and COUNT are different variables.

Command description

NAME	COMMAND	OPERANDS
[name]	{<MULTIPLY> / MPY}	res,m1,m2[,m3,···]

WHERE

res = result; fixed or floating-point symbolic reference.

m1 = multiplicand; fixed or floating-point symbolic reference or constant.

m2 = multiplier; fixed or floating-point symbolic reference or constant. Similarly for m3,m4,···

Function

The function of the "Multiply" command is to perform a multiplication and place the result in the location symbolically represented by the first operand. Any number of factors may be multiplied together and if the "result" reference appears as one of the factors, the old value will be used for the calculation. Full mixed-mode arithmetic is allowed.

Example

1. Suppose we want to cube the number in the location whose symbolic reference is V1, convert the number to floating point, and place the result into a location with the name #CUBE. The following command could be used:

```
<MULTIPLY>  #CUBE,V1,V1,V1
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<DIVIDE> / DIV}	quotient,remain,d1,d2

WHERE

quotient = fixed or floating-point symbolic reference for quotient.

remain = fixed or floating-point symbolic reference for the remainder.

d1 = dividend; fixed or floating-point symbolic reference or constant.

d2 = divisor; fixed or floating-point symbolic reference or constant.

Function

The function of the "Divide" command is to perform a division and place the results in the locations specified by the symbolic references. Full mixed-mode arithmetic may be used; however, if the quotient is floating-point, there will be no remainder term. If a remainder is specified, it will be set to zero. If the symbolic reference for either the quotient or remainder appear as factors for the division, the old value will be used.

Example

Suppose we want to divide the quantity in location #AMOUNT in half; the following command could be used:

```
<DIVIDE>  #AMOUNT,#AMOUNT,2
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<SET VALUE> / SET}	var,val
WHERE		
var = variable to be set; fixed or floating-point symbolic reference or constant. val = value; fixed or floating-point symbolic reference or constant.		

Function

The function of the "Set Value" command is to set the value of some variable to a specified amount.

Example

Suppose we want to initialize the value of the variable symbolically represented by the name AMOUNT to a fixed-point quantity of seventeen. The following command would be used:

```
<SET VALUE>  AMOUNT,17
```

If the value seventeen had already been assigned to another variable, say COUNT, the following command could be used:

```
<SET VALUE>  AMOUNT,COUNT
```

If a floating-point seventeen had already been assigned to the variable named #COUNT, the following command could be used to perform the floating-point to fixed-point conversion prior to the new assignment;

```
<SET VALUE>  AMOUNT,#COUNT
```

APPENDIX G

MESSAGE HANDLING COMMANDS

Formats

In most cases when messages are sent, they will be in a standard format. The text of the message will be edited from three parts. The first part is only present if the message requires a reply; it consists of the reply serial number enclosed in corner brackets. Messages which require a reply are serialized starting with zero then incremented by one for each new message until nine is reached, then the next is serialized with zero again. The purpose for the serial is to allow more than one message requiring a reply to be outstanding for any specific player. When the player replies, he must also precede his reply with the serial to identify it. The second part of the text is the actual message requested by the Control Program. The last part is the numeric content of a variable location which the Control Program can request if required.

In addition to the above editing, all messages which are typed at the player's terminal are preceded with the clock-time that they are sent. This clock-time is the same as that which is stored in the IOCB for this message and it is available to the Control Program when the reply is received. The following is a typical message and its reply:

```
1316:12  <2> WHAT IS NEW SPEED AFTER 1330
<2>  12
```

Operands

Many of the operands that appear in the message handling commands are common to some or all of the commands; hence, they will be described in some detail here.

Message text. The "message text" operand is always coded as a literal constant, i.e., it is coded with the actual message that is to be sent enclosed in apostrophe quotes. If an apostrophe is required within the text of the message, it is coded as a double apostrophe.

Message variable. The "message variable" operand is provided to allow some variable quantity to be edited into the text of the message. It is coded as a fixed or floating-point symbolic reference.

Player-unit designations. The "player-unit" operand is coded to indicate which player is to receive a message. The umpire has the designation zero, all other players have a number in the range (1,11). When the message is to go to several players it may be coded in the list form. In this form the list of players is coded in parenthesis separated by commas, e.g., (0,2,4,5).

Reply event. When a reply is required, it is handled by the Control Program as an event. An event routine is coded assuming that the numeric reply is in modifier three at the time it is executed. Modifier one contains the time that the message was sent and modifier two contains the time that the message reply was received.

Boolean switches. When the "boolean switches" operand is coded in messages requiring replies, the switch settings will be passed to the event routine when the reply is received. In this manner the event routine can identify different messages.

Command description

NAME	COMMAND	OPERANDS
[name]	{<MESSAGE TO LOG> / MSGL}	'mes'[,vl]
WHERE		
mes = message text; literal constant. vl = message variable; fixed or floating-point symbolic reference or constant.		

Function

The function of the "Message to Log" command is to make one entry in the running game log. The message variable, if coded, will be inserted after the text of the message.

Example

Suppose we desire to construct and enter into the log the following message: "FIRST DETECTION MADE BY UNIT7". Suppose, further, that the seven in the text of the message must come from a variable symbolically represented as VU2. The following command could be issued:

```
<MESSAGE TO LOG>  'FIRST DETECTION MADE BY UNIT',VU2
```

When the message is entered into the log, the time will be edited, with both game-time and clock-time, into the entry. The following represents the way the log entry will be made:

```
1316:21  8593  FIRST DETECTION MADE BY UNIT 7
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<MESSAGE TO UMPIRE> / MSGU}	'mes'[,vl]
[name]	{<MESSAGE TO PLAYER> / MSGP}	{(ul,···) / ul},'mes'[,vl]

WHERE

mes = message text; literal constant.

vl = message variable; fixed or floating-point symbolic reference or constant.

ul = player identification; fixed-point symbolic reference or constant. Similarly for u2,···

Function

The function of the "Message to Umpire" command is to send the text of the message, as designated in the first operand to the umpire. It will be assumed that no reply is required. The message variable, if coded, will be inserted after the text of the message. The function of the "Message to Player" command is essentially the same as the "Message to Umpire" command. The message will be sent to the players designated in the list. If it is desired that the umpire be included, zero should be coded in the player list.

Example

The following is an example of a command that will send a message to both the umpire and player three.

```
<MESSAGE TO PLAYER>  (0,3),'SPEED IS',#VEL
```

Command description

NAME	COMMAND	OPERANDS
[name]	{<MESSAGE FOR REPLY> / MSGR}	{(ul,···) / ul} ,'mes',e[,b,vl]

WHERE

ul = player identification; integer in range (0,11); fixed-point symbolic reference or constant. Similarly for other units in list.

mes = message text; literal constant.

e = event designation for reply; integer in range (0,255); fixed-point symbolic reference or constant.

b = boolean switches; binary word with eight bits.

vl = message variable; fixed or floating-point symbolic reference or constant.

Function

The function of the "Message for Reply" command is to send the text of the message, as designated in the first operand, to the player. It is assumed that a reply is required; and, when the reply is sent by the designated player, the event routine designated in the "e" operand will be executed with modifier number three set to the reply that the player sends. In addition, modifier number one will contain the game-time that the message was sent to the player and modifier number two will contain the game-time at which the player responded. The boolean switch settings will also be passed to the event routine for identification purposes.

Example

Suppose we are coding the message used in the example at the beginning of this appendix; the time that is to be inserted into the message text is in the location symbolically represented by TIME. Assume that event routine six is to be executed when the reply is received. The following command could be used:

```
MSGR  3,'WHAT IS NEW SPEED AFTER',6,,TIME
```

This command would cause the message to be typed at the terminal of player three in the following format:

```
1316:12  <2>  WHAT IS NEW SPEED AFTER 1330
```

Assume now that the reply was received from player three at clock-time 1317:27, and that the speed was six. When the reply was sent it would appear in the following form:

```
<2>  6
```

As a result of this reply, the system would transfer control to event routine number six with modifier one set to the game-time associated with 1316:13; modifier two set to the game-time associated with 1317:27 and modifier three set to 6.

Command description

NAME	COMMAND	OPERAND
[name]	{<TABULATE> / TAB}	tab,vl[,freq]
[name]	{<COUNT> / CNT}	cnt[,inc]

WHERE

tab = table designation; integer in range (0,255); fixed-point symbolic reference or constant.

vl = variable; fixed or floating-point symbolic reference or constant.

freq = frequency; integer; fixed-point symbolic reference or constant; defaults to 1.

cnt = counter designation; integer in range (0,255); fixed-point symbolic reference or constant.

inc = increment; integer; fixed-point symbolic reference or constant; defaults to 1.

Function

The function of the "Tabulate" command is to tabulate the variable specified in the "vl" operand in the table designated; the frequency tabulated will be that indicated in the "freq" operand. If this operand is not coded, a frequency of one will be assumed. The "Count" command will cause the counter designated in the "cnt" operand to be incremented by the amount specified.

APPENDIX H

MACHINE CONFIGURATION

This proposal for the Real Time Gaming System is made with the goal of implementation on the computing machinery available at the Naval Postgraduate School. The central processor is an International Business Machines System/360, Model 67-2 with a core storage capacity of 524,288 bytes. Tele-processing is accomplished through a 2702 Transmission Control Unit with a capacity of twelve lines. The remote terminals are 2741 Communication Terminals linked to the central processor by the 2702. Eight 2311 Disk Drive Units and a 2301 Drum are available for auxiliary storage.

The operating system, under which the Real Time Gaming System is designed to run, is the MVT option of OS/360. The input/output is controlled using the Conversational Terminal Access Method (CTAM) developed by the Columbia University Computing Center.

It is anticipated that the system could be implemented on any IBM System/360 with the standard and floating-point instruction sets if a tele-processing capability were available for that system.

APPENDIX I

SAMPLE GAME

The following is a sample of the Control Program for a simple game. The details of the coding are not explained, but the game is straightforward and the programming should be clear. The game itself was not chosen for its elegance, but rather because it uses most of the features of the system.

The object of the game is to test the reaction time of the players in responding accurately to a request from the umpire. The umpire sends a number to the players. They must guess the largest integer in the square root of the number (the answer to 67 would be 8) and respond. The first player to respond is the only one that can score. He receives ten points for a correct answer and loses five points for an incorrect answer. When the umpire wants to stop the game he simply sends a negative number. In no case will the player be allowed more than ten seconds to make his response. The coding provided is not necessarily the best way to program the game; however, it illustrates the details of the programming and shows what a typical Control Program might look like.

SAMPLE PROGRAM

```
NAME        COMMAND                 OPERAND(S)
            <SET TIME>              0
            <START CLOCK>
            <MESSAGE TO PLAYER>     (0,1,2),'READY TO START'
            <NEW FORCE>             1,S
            <NEW FORCE>             2,S
BRP1        <SET SWITCH OFF>        1,(1)
            <SET SWITCH OFF>        2,(1)
            <SET SWITCH OFF>        G,(1,2)
            <MESSAGE FOR REPLY>     (0),'SEND TEST NUMBER',5
DP1         <DO PERIODICALLY>       ,1
            <BRANCH ALL ON>         G,(1,2),BRP2
            <RETURN TO MONITOR>
BRP2        <CEASE>                 DP1
            <SUBTRACT>              $T7,$T1,$T2
            <BRANCH ON VALUE>       $T7,BRP5,,BRP3
            <MESSAGE TO PLAYER>     (0,1,2),'TIE ON TIME, NO SCORE
                                    THIS ROUND'
            <BRANCH>                BRP1
BRP3        <BRANCH ALL ON>         1,(1),BRP4
            <MESSAGE TO PLAYER>     (0,1,2),'PLAYER 2 RECEIVES 10
                                    POINTS'
            <COUNT>                 C2,10
            <BRANCH>                BRP1
BRP4        <MESSAGE TO PLAYER>     (0,1,2),'PLAYER 2 LOSES 5
                                    POINTS'
            <COUNT>                 C],-5
            <BRANCH>                BRP1
```

```
BRP5    <BRANCH ALL ON>          2,(1),BRP6
        <MESSAGE TO PLAYER>      (0,1,2),'PLAYER 1 RECEIVES 10
                                 POINTS'
        <COUNT>                  C1,10
        <BRANCH>                 BRP1
BRP6    <MESSAGE TO PLAYER>      (0,1,2),'PLAYER 1 LOSES 5
                                 POINTS'
        <COUNT>                  C1,-5
        <BRANCH>                 BRP1
        <DEFINE EVENT ROUTINE>   1
        <MESSAGE TO PLAYER>      1,'YOU RAN OVERTIME'
        <MESSAGE TO UMPIRE>      'PLAYER 1 RAN OVERTIME'
        <SET SWITCH ON>          1,(1)
        <END EVENT ROUTINE>
        <DEFINE EVENT ROUTINE>   2
        <MESSAGE TO PLAYER>      2,'YOU RAN OVERTIME'
        <MESSAGE TO UMPIRE>      'PLAYER 2 RAN OVERTIME'
        <SET SWITCH ON>          2,(1)
        <END EVENT ROUTINE>
        <DEFINE EVENT ROUTINE>   3
        <FORTRAN>                $M1,$M2,$M3,#$T10,$T1,$T11
        $T1=$M2-$M1
        $T11=SQRT(#$T10)
        IF($T11.EQ.$M3) GO TO 1
        $M3=-1
      1 CONTINUE
        <END FORTRAN>
        <CANCEL EVENT>           1,$T8
        <SET SWITCH ON>          G,(1)
```

```
        <BRANCH ON VALUE>          $M3,,BRP7,BRP7
        <SET SWITCH ON>            1,(1)
        <BRANCH>                   BRP8
BRP7    <MESSAGE TO PLAYER>        1,'YOUR ANSWER WAS CORRECT;
                                   YOUR TIME WAS',$T1
        <MESSAGE TO UMPIRE>        'PLAYER 1 RESPONDED WITH
                                   CORRECT ANSWER IN TIME',$T1
        <EVENT EXIT>
BRP8    <MESSAGE TO PLAYER>        1,'YOUR ANSWER WAS INCORRECT;
                                   CORRECT ANSWER WAS',$T11
        <MESSAGE TO UMPIRE>        'PLAYER 1 RESPONDED INCORRECTLY'
        <END EVENT ROUTINE>
        <DEFINE EVENT ROUTINE>     4
        <FORTRAN>                  $M1,$M2,$M3,#$T10,$T2,$T12
        $T2=$M2-$M1
        $T12=SQRT(#$T10)
        IF($T12.EQ.$M3) GO TO 1
        $M3=-1
      1 CONTINUE
        <END FORTRAN>
        <CANCEL EVENT>             2,$T8
        <SET SWITCH ON>            G,(2)
        <BRANCH ON VALUE>          $M3,,BRP9,BRP9
        <SET SWITCH ON>            2,(1)
        <BRANCH>                   BRP10
BRP9    <MESSAGE TO PLAYER>        2,'YOUR ANSWER WAS CORRECT;
                                   YOUR TIME WAS',$T2
        <MESSAGE TO UMPIRE>        'PLAYER 2 RESPONDED WITH
                                   CORRECT ANSWER IN TIME',$T2
        <EVENT EXIT>
BRP10   <MESSAGE TO PLAYER>        2,'YOUR ANSWER WAS INCORRECT;
                                   CORRECT ANSWER WAS',T12
```

```
        <MESSAGE TO UMPIRE>      'PLAYER 2 RESPONDED INCORRECTLY'

        <END EVENT ROUTINE>

        <DEFINE EVENT ROUTINE>   5

        <BRANCH ON VALUE>        $M3,BRP11

        <SET VALUE>              #$T10,$M3

        <MESSAGE TO PLAYER>      (1,2),'WHAT IS LARGEST INTEGER
                                 IN THE SQUARE ROOT OF',$M3
        <PRESENT TIME>           $T9

        <ADD>                    $T8,$T9,10

        <STORE EVENT>            1,$T8

        <STORE EVENT>            2,$T8

        <EVENT EXIT>

BRP11   <END OF GAME>

        <END EVENT ROUTINE>
```

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2.	Library Naval Postgraduate School Monterey, California 93940	2
3.	Director, Systems Analysis Division OP96 Office of the Chief of Naval Operations Washington, D. C. 20350	1
4.	Prof. G. L. Barksdale, Jr. Department of Mathematics Naval Postgraduate School Monterey, California 93940	2
5.	Prof. R. H. Shudde Department of Operations Analysis Naval Postgraduate School Monterey, California 93940	1
6.	Lt. Edward A. Singer Box 1360, Naval Postgraduate School Monterey, California 93940	15
7.	Department of Operations Analysis Naval Postgraduate School Monterey, California 93940	1
8.	Prof. A. F. Andrus Department of Operations Analysis Naval Postgraduate School Monterey, California 93940	1
9.	Prof. G. Heidorn Department of Operations Analysis Naval Postgraduate School Monterey, California 93940	1
10.	Prof. D. G. Williams Computing Center Naval Postgraduate School Monterey, California 93940	1

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)
Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3 REPORT TITLE

A REAL TIME GAMING SYSTEM

4 DESCRIPTIVE NOTES (Type of report and, inclusive dates)
Thesis

5 AUTHOR(S) (First name, middle initial, last name)
SINGER, EDWARD ANTHONY, JR., Lieutenant, USN

6 REPORT DATE
December, 1968

7a. TOTAL NO. OF PAGES
139

7b. NO. OF REFS
6

8a. CONTRACT OR GRANT NO

b. PROJECT NO

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10 DISTRIBUTION STATEMENT

Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Naval Postgraduate School
Monterey, California 93940

13 ABSTRACT

A system is proposed which will support computer gaming in real-time. This system will, when combined with the user's Control Program, monitor all of the functions necessary to provide real-time man/machine interaction with the game. The formal definition of a programming language (RTGS Control Program Command Language) is given; this language, supplemented by Fortran IV and IBM OS/360 Assembler Language is used for coding the user's Control Program. Plans for implementation on an IBM System/360 Model 67 are discussed and a sample program is given.

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
GAME THEORY						
WARGAMES						
REAL TIME						
COMPUTER OPERATING SYSTEMS						
COMPUTER MONITOR SYSTEMS						
COMPILER						
TELE-PROCESSING						